The muMATH/muSIMP-80™ Symbolic Mathematics System for the Apple II with SoftCard

Reference Manual

muMATH/muSIMP-80 is distributed exclusively by
Microsoft
10800 N.E. Eighth, Suite 819
Bellevue, WA 98004

8712-210-014

INTRODUCTION

muMATH-80 is a fully interactive **Symbolic Math System** that efficiently and accurately performs true algebraic and analytic operations. These operations are utterly beyond the built-in facilities of traditional scientific programming languages such as APL, BASIC, FORTRAN, PASCAL, or PL/I. Unlike those languages and unlike typical scientific subroutine libraries written for them, muMATH can evaluate and simplify expressions containing variables that have not been assigned numeric values.

For example, muMATH can automatically expand expressions over a common denominator, employ trigonometric identities to simplify expressions, and symbolically integrate expressions exactly. Operations such as matrix inverses and matrix products can be computed by muMATH even when the matrices contain non-numeric entries. Unassigned variables in the data are carried along algebraically just as is done in algebra, trigonometry, and calculus courses. Moreover, the arithmetic performed on numerical coefficients of such variables is exact rational arithmetic for numbers up to 600 decimal digits.

muSIMP-80 is the general-purpose high-level **programming language** in which muMATH is implemented. muSIMP is designed especially for implementing computer algebra systems. It is provided along with muMATH so that users who wish to do so can use muSIMP to extend muMATH or to implement other Artificial Intelligence applications. However, most users should find the built-in features of muMATH sufficiently powerful to handle the majority of their mathematical problems.

muSIMP/muMATH was designed and implemented by David Stoutemyer and Albert Rich of The Soft Warehouse, Honolulu, Hawaii. The project was begun in 1976 by Rich with the design and coding of a LISP language interpreter on an 8080 based microcomputer. Convinced of the power and utility of symbolic mathematics, Stoutemyer collaborated with Rich to greatly upgrade the interpreter in both speed and numerical ability. The muSIMP **surface language** was devised during this period to give the user the power of an applicative language like LISP but with a more natural syntax. Finally, using muSIMP, the muMATH Symbolic Math System was written making it the first such system available on microcomputers.

TABLE OF CONTENTS

1. HOW TO USE THIS MANUAL

We expect that muSIMP or muMATH will attract people having a wide variety of computer experience ranging from none to extensive. Accordingly:

1. In order to make this documentation understandable to computer novices, we have avoided as much as possible the customary computer jargon that was not deemed indispensable.

2. In order to help those who are unfamiliar with using computer terminals and the types of systems on which muSIMP/muMATH runs, Section 4 contains general information designed to ease a first encounter with such hardware and software.

3. Terminology that we think will be new to many readers is generally explained, at least partially, at or near its first occurrence. Such terms are usually printed **boldface** in sentences that significantly contribute toward their explanation, in order to emphasize that the terminology will be used later without explanation, or in order to make the sentence easier to find from the entry in the index. We also use boldface for section titles and occasional emphasis of nontechnical words.

4. In order to avoid distracting digressions to explain terminology that we expect to be familiar to most but not all readers, such terms are <u>underlined</u> at first occurrence in order to indicate that they are explained in the Glossary.

5. If an unfamiliar term is encountered, the index or glossary should lead to the definition. If the term is not indexed, check other personal computing books or ask a more experienced person. Finally, if all else fails, contact your muMATH distributor for an explanation.

We expect readers will refer to this manual for a great variety of reasons over time. Thus, here is a quick guide for answers to some of the most common questions:

	Question	Answer
1.	What are muSIMP & muMATH?	See Section 2.
2.	Will muSIMP/muMATH run on my computer?	See Section 3.
3.	What kind of computer should I obtain to run muSIMP/muMATH?	See Section 3.
4.	Do I know enough about programming in general and using my computer in particular?	See sections 2 & 4.

5. Do I know enough mathematics?	See Sections 2 & 8.
6. What do I do first upon receiving muSIMP/muMATH by mail-order or from a dealer?	See Sections 4 - 6.
7. How do I start muSIMP/muMATH running?	See Sections 5 & 6.
8. How do I save a muMATH environment to achieve faster subsequent startup or to continue a dialog at a later time?	See Section 7.
9. How do I learn to use muMATH?	See sections 6 & 8.
10. Having learned to use muMATH, how do I look up a specific fact about one of the specific muMATH packages?	See the appropriate subsection in 9, or consult the Index.
11. Having learned muMATH calculator mode, how do I learn muSIMP in order to extend muMATH or implement other applications?	See Section 8.
12. Having learned to use muSIMP, how do I look up a specific fact about one of the muSIMP features?	See appropriate subsection in 12 or 13, or consult the Index.
13. How do I learn more about computer algebra and artificial intelligence programming in general?	See sections 10 & 14.
14. How can I use muSIMP/muMATH in an educational role?	See Section 11.

We have tried to anticipate all of the problems beginners might have with hardware, operating systems, muMATH and muSIMP. We have almost certainly overlooked some, and we would appreciate learning of them so that we can improve this manual.

Meanwhile, we urge you to work together with friends. Even if two people are equally inexperienced with a computer or software system, together they can often figure out things that neither could alone. Occasionally rereading this manual may reveal subtleties that were overlooked earlier.

2. AN OVERVIEW OF muMATH & muSIMP

What are muMATH and muSIMP?

First, muMATH: Many who have not yet used a traditional scientific programming language such as ALGOL, APL, BASIC, FORTRAN, PASCAL, or PL/I may be surprised and disappointed to learn that their built-in mathematical capabilities are essentially limited to arithmetic: Only numbers can be substituted into the formulas expressed in these languages when the program runs.

In contrast, variables in muMATH formulas do not require numerical values when they are evaluated: Variables that do not have numerical values are carried along algebraically in the result, just as is done in algebra, trigonometry and calculus courses -- muMATH can actually transform formulas into other equivalent formulas, either automatically or at the explicit request of the user.

Virtually no knowledge of computer programming is required to use muMATH in this **calculator mode**, beyond the conventions that:

1. an asterisk, *, is used to denote multiplication,

2. an upward-pointing arrow, ^, is used to denote exponentiation (i.e. "raising an expression to a power"),

3. the end of each expression entered by the user is signaled by typing a semicolon followed by a carriage return.

For example, suppose that a user wants to expand and simplify the mathematical expression

$$2y(y^2 - z) + 2z(y + z)$$

The user merely enters

```
2*Y*(Y^2 - Z) + 2*Z*(Y + Z);
```

and within a second or so muMATH displays the expanded equivalent

```
2*Y^3 + 2*Z^2
```

Other algebraic transformations include expansion of expressions over a common denominator and partial factoring. If desired, simplified expressions can be saved for use in subsequent expressions, thus permitting complicated sequences of interrelated operations.

muMATH can also accept genuine equations as expressions, independently simplifying the two sides. Equations can be added, multiplied, etc. in order to derive solutions stepwise. There is a built-in function named SOLVE that automates the process. For example,

to solve the equation

$$x(3+x^2) = 4x(1+c^2)-x$$

for x in terms of c, the following expression is entered:

```
SOLVE ( X*(3+X^2)==4*X*(1+C^2)-X , X);
```

In a few seconds muMATH displays the solution set

```
{X == -2*C,
 X == 2*C,
 X == 0}
```

muMATH also supports arrays, matrix algebra and determinants, with nonnumeric entries permitted. For example, to find the inverse of the matrix

$$\begin{bmatrix} 1 & 2yz^2 \\ y & 2 \end{bmatrix}$$

the user merely enters the expression

```
{[1, 2*Y*Z^2],
 [Y,     2  ]} ^ -1;
```

and in a few seconds muMATH displays the matrix

```
{[1+2*Y^2*Z^2/(2-2*Y^2*Z^2), -2*Y*Z^2/(2-2*Y^2*Z^2)],
 [-Y/(2-2*Y^2*Z^2), -1/(2-2*Y^2*Z^2)]}
```

muMATH also can perform closed-form summation even for nonnumeric summation limits. For example, to determine a closed form for

$$\sum_{j=0}^{n-1} (c^j + j^2)$$

in terms of c and n, the user enters

```
SIGMA (C^J + J^2,  J,  0,  N-1);
```

and in less than a minute muMATH displays

```
(-36-6*N+6*C*N-18*C*N^2+12*C*N^3+36*C^N+18*N^2-12*N^3) / (-36+36*C)
```

muMATH also has an extensive array of automatic and optional transformations for logarithmic, exponential, and trigonometric expressions. For example, muMATH can simplify the expression

$$\sin(2y)\ (4\cos^3 x - \cos(3x)) + (\cos(x+y+\pi) - \cos(x-y))\sin y$$

into

$$4 \sin y \cos x \cos y$$

in a less than half a minute.

muMATH also has calculus capabilities. For example, to evaluate the integral

$$\int (c\,x^2 + x \sin(x^2))\,dx$$

the user enters

```
INT (C*X^2 + X*SIN(X^2), X);
```

and in a few seconds muMATH responds

```
C*X^3/3 - COS(X^2)/2
```

muMATH also knows how to use differentiation to derive truncated Taylor series expansions. For example, to determine the 5th degree truncated Taylor-series expansion of $e^{\sin x}$, about x=0, the user enters

```
TAYLOR (#E^SIN(X), X, 0, 5);
```

and in less than a minute muMATH responds

```
1 + X + X^2 - X^4/8 - X^5/15
```

muMATH even knows L'Hospital's rule for determining limits of indeterminate forms. For example, to determine

$$\lim_{x \to 0} \frac{a^x - a^{\sin x}}{x^3}$$

the user enters

```
LIM ((A^X - A^SIN(X)) / X^3, X, 0);
```

and in about a minute muMATH responds

```
LN(A) / 6
```

muMATH arithmetic, by the way, is exact rational arithmetic. For example, when the user enters

```
1/2 + 1/3
```

the result is exactly 5/6, without roundoff error. As another example, when the user wishes to compute the tenth power of ten factorial, the entry

```
10! ^ 10;
```

yields the exact result

```
3959408661224251932438755707826684577630388224000000000000000000000
```

within about a second.

The size of expressions that muMATH can accommodate is limited only by the total available RAM memory. Calculations involving over a hundred terms having coefficients of up to several hundred digits are quite possible.

As illustrated by the above examples, muMATH has broad mathematical capabilities spanning the content of algebra through sophomore calculus. However, muMATH is modular so that users need not know higher-level math in order to use the lower-level capabilities such as algebra or exact rational arithmetic.

The above examples show that muMATH is as convenient and interactive as using a pocket calculator. Expressions are transformed and simplified as they are entered. Thus using muMATH in this **calculator mode** eliminates the need to write and submit long programs. The user is free to experiment by entering and manipulating formulas and can immediately see the results. Mistakes are discovered quickly so that the formula can be reentered without having to rerun a program. Seeing each result as it is computed often suggests new avenues to explore that couldn't have been forseen if the entire sequence of calculations had to be preplanned. It is somewhat similar to the difference between corresponding by phone and by mail: one is a dialogue whereas the other is a sequence of alternating monologues separated by long delays.

The built-in facilities of muMATH so thoroughly span mathematics through sophomore calculus that we expect many users will be content to use only calculator mode. However, muMATH is written in a high-level language called muSIMP, which was specially designed for the task. muSIMP is provided along with muMATH, so that users can extend the built-in capabilities as desired. Moreover, the muSIMP program files defining muMATH are provided in their original form with muMATH so that users can modify them as desired or employ them as models for their extensions.

To this end, muMATH comes with a set of lessons that explain how to program in muSIMP. However, unlike the calculator mode, use of the **programming mode** requires previous programming experience for most users — no language in particular, just a certain amount of programming sophistication. The reason is simply that despite the substantial bulk of this manual, the programming lesson portions are rather brief for a first exposure to computer programming.

To satisfy the curiosity of those with previous programming experience, here, without further explanation at this time, is a definition of a muSIMP function that returns the n^{th} **Fibonacci** number,

which is defined by the relations

$$F(0) = 1,$$
$$F(1) = 1,$$
$$F(n) = F(n-1) + F(n-2), \qquad \text{for } n>1.$$

```
FUNCTION F (N),
   WHEN  N=0 OR N=1,  1  EXIT,
   F(N-1) + F(N-2),
ENDFUN;
```

Even without delving into details here, it is evident that the function definition closely parallels the mathematical definition. This definition can be entered interactively at the terminal or read from a file prepared by another program called a text editor. The function is immediately available for use in either case. For example, if the user then enters

```
F(5);
```

The displayed result is 8.

Computer algebra grew out of the field of **artificial intelligence,** so it is not surprising that a programming language which is well suited to one is well suited to the other. muSIMP is in fact a good choice for all artificial intelligence applications, and the programming mode lessons teach general techniques that are useful throughout the field.

3. PREREQUISITE HARDWARE & SOFTWARE

muSIMP, hence muMATH, currently runs only on certain microcomputers employing the Intel 8080, Intel 8085, or Zilog Z80 microprocessor CPU chips. Moreover:

1. The computer must provide a minimum of about 24 **kilobytes** of **RAM** memory for user programs, excluding any RAM that is occupied by the computer's operating system.

2. The operating system must be one of several specific disk operating systems.

3. The software is distributed only on some of the various floppy disk formats that are used.

4. Different formats are available from different sources.

Here are the specific computers and operating systems for which muSIMP/muMATH are currently available:

COMPUTER	OPERATING SYSTEM
Radio Shack TRS-80, Model I, Level II	TRSDOS
Cromemco Z1, Z2 & Z3	CDOS
IMSAI VDP	IMDOS
Apple][with the Microsoft Z80 SoftCard	CP/M
Heath H89 with the Magnolia Microsystems of Seattle, piggy-back board	CP/M or MP/M
Any of dozens of other brands that are based on an 8080, 8085 or Z80, and have at least 24 K of RAM contiguous from location 100 hex. These include the North Star Horizon, Vector MZ or 1+, iCOM, Exidy Sorcerer, ISC Intecolor, Intel MDS, IMSAI 8080, Ohio Scientific C3, Intertec Superbrain, Onyx, Altair, Altos, Dynabyte, and Research Machines.	CP/M or MP/M

The Soft Warehouse has also implemented a versatile LISP pseudo-code interpreter for these machines distributed under the tradename muLISP.

By far the most common disk format is 8-inch single-density "standard IBM" format, but regretably, there are about a half dozen other 8-inch formats and over fifty 5 1/4 inch formats — too numerous and rapidly changing to list here.

MICROSOFT, of 10800 N.E. Eighth, Suite 819, Bellevue, Washington 98004, U.S.A. has an exclusive license to distribute and sub-license dealerships of these **SOFT WAREHOUSE** products. Dealership and OEM inquires concerning muMATH or muLISP should be made directly to Microsoft. Their phone number is (206) 455-8080.

These products are currently available to end-users in a wide range of diskette formats from various mail-order software houses. The following is a list of some of these distributors:

1. **MICROSOFT CONSUMER PRODUCTS**, 400 108th Ave. N.E., Suite 200, Bellevue, WA, 98004, U.S.A. Phone: (206) 454-1315. They distribute muMATH and muLISP for the **TRS-80** and **APPLE][** microcomputers, and for CP/M based systems using standard IBM formatted 8" diskettes.

2. **LIFEBOAT ASSOCIATES**, 1651 Third Avenue, New York, NY, 10028, U.S.A. Phone: (212) 860-0300. They are a large mail-order distributor of CP/M and CP/M based software in a wide variety of formats.

3. **DISCOUNT SOFTWARE**, The Discount Software Group, 1610 Argyle Ave., Bldg. 101, Los Angeles, CA, 90028, U.S.A. Phone: (213) 666-7677.

4. **TSE/HARDSIDE**, 6 South St., Milford, NH, 03055, U.S.A, Phone: (603) 673-5144 or toll free 1-800-258-1790.

5. **MICROHOUSE**, 511 North New Street, Bethlehem, PA, 18018, U.S.A. Phone: (215) 868-8219.

If you are in need of a muMATH compatible operating system, the easiest way to get CP/M on your system is through the manufacturer of your computer hardware and/or disk drives. When obtained in this way little or no customizing will be necessary to install the operating system. Alternative sources for CP/M include **Lifeboat Associates** and **Digital Research** at P.O. Box 579, 801 Lighthouse Avenue, Pacific Grove, California, 93950, U.S.A.

4. EFFECTIVE USE OF YOUR COMPUTER

There is a certain amount of work required to learn to use any computer system -- especially if it is your first encounter with computers. It is not so much a matter of understanding profoundly difficult concepts; rather, it involves learning a fair number of procedural details. These details should be documented in the hardware and software reference manuals, but documentation has a way of becoming separated from a computer system. This is particularly true when more than one person is using a computer. Worse yet, these manuals tend to overwhelm beginners with the details of every single feature, presented in an order more convenient for reference than for initial learning.

To alleviate this problem, the documentation for muMATH and muSIMP includes both this extensive reference manual as well as on-line tutorials. These tutorials are in the form of interactive lessons files provided with the system. Unfortunately describing the numerous and rapidly changing computer hardware configurations is a much more difficult task. Thus when initially learning how to use your computer, personal help from a knowledgable person is the fastest and least painful means of accomplishing this. If you lack access to such an experienced person and/or tutorial material for the computer hardware and operating system, this section is an attempt to present sufficient information necessary to use muSIMP/muMATH effectively.

4.1 TERMINALS

There are hundreds of different kinds of terminals with differing keyboards and attributes. Here are some of the typical features relevant to running muMATH:

1. The **power switch** is often on the back, side, or underside of the terminal. This sometimes inconvenient position was probably chosen by the manufacture to minimize the chance of accidentally shutting off the terminal and to isolate the line-voltage from more delicate components.

2. The terminal may have a **communicate switch** or key marked LOCAL versus REMOTE, or LINE. The LOCAL position disconnects the terminal from the computer, for stand-alone use similar to a typewriter. Thus the REMOTE position is the appropriate setting for communicating with a computer.

3. The terminal may have a **parity enable switch** usually marked PARITY ON versus PARITY OFF, and/or a switch marked PARITY ODD versus PARITY EVEN. Most microcomputer systems will work fine with PARITY OFF, making the ODD/EVEN parity switch ineffectual. However, if PARITY ON must be used, the proper position for the ODD/EVEN switch can be determined by experimentation.

4. If the terminal has a **duplex mode switch,** usually marked HALF-DUPLEX versus FULL-DUPLEX, it should be used in the FULL-DUPLEX mode.

5. The terminal's **data transmission rate** selector switches allow the terminal to operate at various **BAUD** rates. This rate is the number of bits per second that are sent and/or received by the terminal. Ten bits are required for each character transmitted. Modern terminals generally provide a generous of selection of rates from 110 to 19,200 BAUD. The rate chosen must match that of the computer. If you have the ability to set both the computer's and the terminal's BAUD rate, pick the highest possible rate since it enhances the interactive nature of the computer system.

6. Many terminals have a useful switch marked **CAPS-lock** or perhaps UPPER-case versus LOWER-case-enable. In the upper-case position all characters transmitted by the terminal will be upper-case characters. Unlike typewriter SHIFT-lock keys, CAPS-lock does not affect non-alphabetic keys. Since some terminals can only transmit upper-case characters, muMATH function and variable names contain only capital letters. It is advisable to use CAPS-lock if you wish to make the names in your programs accessible from all kinds of terminals. Note that some terminals display lower-case letters merely as smaller capitals, so you may be deluded into thinking you are typing capitals when you are not.

7. Many terminals have a **control** key which is usually marked either CTRL or CNTL. Typing a character while depressing the **CTRL** key sends the computer a special code for which there is no corresponding displayable character, as if the control key were a special sort of shift key. The names of these codes are sometimes abbreviated above the corresponding non-control characters on the keys they share. For example, CTRL-G rings a "bell" on most terminals, so the abbreviation "BEL" may appear on the G key. A common source of error is to accidently get such an undisplayable character inserted in the middle of a program. Since the troublesome character is invisible, its presence is unlikely to be noticed. To alleviate this problem, when a control character is typed, many systems print a special character such as "^", followed by the corresponding non-control character, in order to indicate that the control character has been received.

8. Unlike typewriters, most terminals have separate **LINEFEED** and **CARRIAGE RETURN** keys, respectively marked something like LF and either RET, RETURN, or ENTER. On most systems, the user generally types only carriage returns, and the system automatically generates corresponding linefeeds too. A common source of errors is to type a linefeed instead of a carriage return when the latter is required. For example, a linefeed typed while in column 1 looks the same as a carriage return followed by a linefeed typed from column 1. The symptom of this blunder is often a lack of the activity expected of the computer following a carriage return because the program is still awaiting a carriage return.

9. Some terminals have one or more keys designed to simplify the entering of commonly used control characters. These keys are marked with the name of control character. Examples include the **ESC**ape key, and/or the **ALT**mode key. Some of these keys are used by muSIMP to interrupt a computation in progress.

10. **CRT** (Cathode Ray Tube) terminals indicate the current position on the screen by using a cursor indicated by a block of light. The terminal may have special keys or control-keys marked with arrows to control the cursor's movement. The operating system or certain programs that run on the operating system may not have been adapted to respond to them, because terminals' cursor control conventions vary widely among manufacturers.

11. Some terminals may not provide for the full ASCII characters set. For example, the characters

    ] [} { \

    used in the muMATH array and matrix packages may not appear on any key. If there is no way to enter and/or display these characters, use a text editor to replace them with other characters or names in the array and matrix source files.

4.2 ACOUSTIC COUPLERS

Some computer installations may use an acoustic coupler to provide for communication between a terminal and a computer via the phone system. If this is the case, the power, duplex, parity, etc., switches on the coupler must be set as described for terminals above. The steps of protocol involved in dialing the computer and logging-in vary widely, so consult a local expert for this information.

4.3 FLOPPY DISKS

Serious programming requires a method of saving user-generated programs on a **mass storage** device so that they do not have to be manually entered for each use. On microcomputers, the most popular methods of mass storage are paper tape, audio-cassette tape, floppy disk, and hard disk. If a user is expected to operate the computer, he/she must learn how to load programs using the medium currently employed. Typical paper tape and audio-cassettes are impractical for programs as large as muMATH, and hard disks are not yet prevalent on microcomputers. Consequently, most muMATH users will be using floppy disks.

Floppy disks are flexible 5 1/4 inch or 8 inch diameter circular plastic disks coated with a magnetizable surface similar to that of magnetic tape. The 5 1/4 inch size is sometimes called a **minifloppy diskette** to distinguish it from the larger size. The disks are permanently enclosed in a square cardboard cover. The plastic disk and cover have a hole in the center for a drive hub. There are usually 1 or 2 smaller holes through the cover next to the center hole, with a corresponding **index hole** through the plastic disk. **Hard sector** disks have additional holes through the disk at the same radius as the index hole to indicate the beginning of each sector within the track. There are usually radial slots on both sides of the cover to provide access for the disk drive's read/write heads.

The notches on the edge of the cover of most floppy disks can be used to protect valuable files on the disk from accidental erasure. Generally, 5 1/4 inch disks are protected when the notch is covered with opaque tape, whereas 8 inch disks are protected when the tape is absent. This **write protect** mechanism reduces the chance of inadver-tently writing over irrecoverable files, such as those on the master copy of muMATH. An adhesive label on one side of the cover distinguishes the two sides of the disk and can be used to identify the contents.

Disk drives have a slot in which to insert a diskette. Once inserted some sort of latch will hold the disk in place. Disks are usually inserted so the radial slot on the cover of the disk will be innermost in the drive. The proper direction for the label to face will soon become evident when an attempt is made to use the system. Although some drives can actually read either side of a disk, usually the manufacturer of the disk guarantees the quality of only one side.

When there is more than one disk drive, one of them is usually designated as the **prime** or **system** drive — often by the letter **A**, or the digit **0** or **1**. Drives usually have a **head loaded light** that is on when the read-write head is in contact with the disk's surface.

Floppy disks have exposed magnetic surfaces, so meticulous care is necessary in order to prevent a loss of your dearly purchased or carefully developed programs. Accordingly:

1. Always store the disk in its paper dust jacket when it is not in a drive. Otherwise, dust from the air will settle on the surface, or dust from the surface that the disk is placed upon will adhere to the exposed undersurface, which may be the side that is recorded even if the label is on the other side. Not only can dust make the disk impossible to read or write, but it can cause harmful abrasion of the drive's read/write head.

2. For transportation and long-term storage, keep the disk and dust jacket in one of the hard plastic boxes designed for that purpose.

3. Buy disk brands that have a good reputation for use in conjunction with your particular drive, even if they cost significantly more.

4. Periodically make backup copies of new files onto other disks, as described later in this section.

5. Keep your floppy disks in a clean environment. For instance do not smoke around disks or expose them to chalk dust.

6. Don't bend disks or sandwich them between non-flat objects.

7. Don't leave disks exposed to direct sunshine, heat, excessive cold, or dampness.

8. Never touch the exposed surface of a disk.

9. Don't turn the computer and/or disk drive ON or OFF when a disk is in the drive with the latch closed.

10. Don't open the drive latch while the drive light is on.

11. Don't submit disks to airline security X-rays, magnetic metal detectors, or magnetic book theft detectors at libraries.

12. Write on disk labels only lightly with a felt-tip pen.

13. Rather than erase information on disk labels, peel the label off or cover it with another.

14. Don't place disks near power-cords, motors or transformers in your computer or terminal.

15. If your disk drives have a tendency to crimp or wear the edge of the central diskettte hole, causing misalignment or slippage, then use the plastic reinforcement rings sold at computer stores or by mail order, as advertised in major computing magazines.

16. Have your drives checked for head alignment according to the schedule recommended by the manufacturer, or do it yourself using a commercially available alignment disk and oscilloscope.

4.4 MONITORS & OPERATING SYSTEMS

Most computers operate under the control of a general-purpose supervisory program. A small supervisory program, usually implemented in ROM memory (Read Only Memory), is called a **monitor.** A large and flexible supervisory program designed to support files and mass storage devices is called an **operating system.** If disk drives are the means of mass storage it is called a **disk operating system** (**DOS**).

Operating systems vary widely, so we can only indicate some of the general principles here that are relevant to running muMATH. With the exception of Radio Shack's TRSDOS^{tm}, all of the operating systems for which muSIMP/muMATH is currently available are quite similar to CP/M so we refer to this family of operating systems as **CP/M-like.** To avoid vague generalities we use CP/M commands as specific examples. However, even TRSDOS has commands analogous to these, so users of non-CP/M operating systems should have no trouble substituting their equivalent commands in our examples. To this end, we suggest that you pencil in the appropriate commands in this manual wherever they differ from those that we describe.

4.5 COLD-STARTING THE COMPUTER

To **cold-start** a computer means to do all the steps necessary to load the operating system from scratch and turn control over to the system. Computers that run muSIMP are started more or less as follows:

1. Turn on the power to the computer, terminal, and disk drives.

2. Insert a properly oriented disk containing the operating system into the prime drive, then close the latch.

3. Initiate the operating system's **bootstrap loader** (i.e. the sequence of steps necessary to load and execute the computer's disk operating system). This procedure varies so widely that it is impossible to provide any detailed instructions. It usually involves forcing the computer to begin execution at a certain memory location. This causes the operating system to be read off the disk into RAM memory and then control is passed to this program. See your computer's documentation for details.

4. When the system has been loaded, the operating system's signon and **prompt** should be displayed on the terminal. This indicates that the system is ready to receive a command from the user. For CP/M this prompt consists of the letter designating the currently **logged** drive, followed by the symbol ">". Thus, since CP/M initially cold starts on drive A, the prompt is

```
A>
```

4.6 THE DOS COMMANDS

CP/M commands are entered by typing the desired command followed by a carriage return. There are two basic types of CP/M commands. **Intrinsic commands** are so named because the programs that execute them reside in the portion of the operating system that is always in memory when one is interacting with CP/M at the **operating system level. Transient commands,** being larger or less frequently needed, are loaded into memory each time they are executed. They can only be executed if a disk contains them as a **COM**mand file. Because of size limitations, it is often necessary to delete some of the more expendable transient commands from disks that contain other large programs such as muSIMP/muMATH.

4.6.1 THE DOS INTRINSIC COMMANDS

The command for changing the drive that is currently logged is simply the letter designating the desired drive followed by a colon. For example, the command

```
B:
```

changes the logged drive from whatever it was to B, resulting in subsequent prompts of

```
B>
```

The command for displaying the directory on the currently-logged drive is

DIR

Each file has a **primary** name of up to 8 characters followed by a period and a **secondary** name of up to 3 characters. The secondary name is also called the **type**, because it is often used to indicate the kind of file: For example, **COM** indicates a machine language program that is a transient command. In general, files can be designated by the letter of the drive they are mounted on, followed by a colon, then the primary name, then a period, then the type. In other words, **file designations** can have the form:

drv:nam.typ

For example, the command file MUSIMP.COM on drive B is designated by the following:

B:MUSIMP.COM

The drive and colon can be omitted when the drive is the currently-logged one.

Many commands have a file designation as an **operand.** For example, the command to **ERA**se a file has the form:

ERA drv:nam.typ

For example, the command

ERA FORTRAN.HEX

will erase the file FORTRAN.HEX from the directory. The erase command should of course be used with great care since it can destroy a great deal of work with little effort. Always make sure the correct disk is used when specifying a file to be erased.

Some commands have more than one operand. For example, the command for **REN**aming a file has the form

REN newdesignation=olddesignation

such as

REN VECTOR.ARI=VECT.ALG

The TYPE command displays the contents of a file on the terminal, which is quite useful for determining the contents of the file. This command has the general form

TYPE filedesignation

Note, however, that COM files and other machine-language files contain **bit patterns** that do not correspond to any displayable character, and that the bit patterns that do display bear no easily discerned relation to what the program is. Thus, using TYPE for such files produces what appears to be gibberish.

If the display from a TYPE command or if any other display is **scrolling** off the top of a CRT screen too fast for comfortable reading, then typing **CTRL-S** causes the display to be temporarily Suspended. Typing another CTRL-S or any other character then permits the output to resume from where it was suspended.

Sometimes after seeing the first few lines of a lengthy output, the user decides not to look at the rest. If so, the remaining output can be aborted by typing any character other than CTRL-S.

Systems with CRT terminals may have a printer attached to the computer for producing **hard copy** printout of what appears on the terminal. If the printer is on and properly interfaced to the computer, typing **CTRL-P** on the terminal when interacting with CP/M will cause susequent output to the terminal to be direct to the printer as well. This will continue until another CTRL-P has been entered.

4.6.2 THE DOS TRANSIENT COMMANDS

An important example of a transient command is the **STAT**us command, which indicates how much available space remains on a disk. The most simple form of the command is merely

STAT

This causes the amount of available space remaining on the disk to be displayed on the terminal. If a file designation is given following STAT, then the space required by that file is also displayed.

MUSIMP itself is a transient command which loads and initiates the muSIMP interpreter. It will be throughly described in later sections.

4.6.3 THE DOS WARM-START

Besides the cold start described in section 4.5, CP/M can be **warm started** by typing **CTRL-C.** This has the effect of reloading the operating system from the currrently logged drive, and updating certain tables in memory that indicate which tracks are available for writing on each mounted disk. Moreover, many programs interpret CTRL-C as the signal to exit the program and return to the **operating system level.**

On some CP/M-like operating systems, attempting to write on a disk that has been inserted since the last warm or cold start may destructively overwrite and scramble information. Thus, **always type CTRL-C immediately after inserting a new disk in a drive.** The only exception to this might be when you are inside a program such as muSIMP and you merely want to **read** a file from another disk. This will work fine; however, we recommend using a write protected disk in this case.

4.7 LINE EDITING

There is no way to modify a CP/M command after the carriage return is typed. Use of a nonexistent command name or operand usually results in a harmless question mark error message followed by a new prompt. However, it is wise to pause for reflection before typing the return on potentially damaging commands such as ERAse. Provided a carriage return has not been entered, CP/M does provide a number of **line-editing** facilities for changing the current line.

For CRT terminals, type **CTRL-X** to erase the entire current line, or type one **CTRL-H** or perhaps a key marked something like "Back Space" or "<-" for each character that you want to erase from the right end of the current line. On many CP/M-like systems the cursor appropriately backspaces, and the affected characters disappear from the screen. On other systems, the treatment is regrettably the same as for **CTRL-U** and **RUB**out or **DEL**ete, as described in the next paragraph.

For hard-copy terminals, type **CTRL-U** to erase the entire current line, or press the key marked either **RUB**out or **DEL**ete once for each character that you desire to erase from the right end of the current line. Since hard-copy terminals can not erase an already printed character, the computer indicates a line deletion by printing a "#" sign at the end of the deleted line and then starting a new line. A character deletion is indicated by the computer's printing of the deleted character a second time and/or printing special characters. More than one or two character deletions on a line leads to a rather confusing appearance. Consequently, type a **CTRL-R** anytime you want a clean copy of the current line **R**etyped on a new line.

The information in sections 4.4-4.7 is all that one has to know about CP/M to use muMATH in calculator mode. Users who wish to proceed beyond that to writing muSIMP programs will also need to learn how to use a **text editor** such as the one named **ED** that is provided with CP/M. Since there is a wide variety of text editors on the market, we do not provide instructions for using an editor. If you do not already know how to use an editor on your system, you must rely on a manual or, preferably, the personal help of a friend. In any event, we suggest not bothering with a text editor unless all of the calculator-mode and programming-mode lessons have been completed and you wish to develop a muSIMP package.

4.8 DISKETTE BACKUP

Before attempting to run muSIMP/muMATH from the master disk(s) obtained from a dealer, a duplicate copy should be made as a safeguard against loss and physical or magnetic damage. The original master disk(s) should then be stored in a safe environment separate from the duplicate **working copy**. As specified in the license agreement, copies are for use only on the originally specified machines, and each copy should bear the following phrase typed or handwritten on an adhesive label:

In order to make a CP/M-compatible disk containing the operating system together with all of the files on a muSIMP/muMATH master disk:

1. The new disk must be **formatted**, meaning the magnetic track and sector boundaries should be properly recorded on the disk.

2. A copy of the operating system matched to the full current memory size of the machine should then be transferred to the disk.

3. Next the files from the master disk should be copied onto the disk. We strongly suggest that you physically write-protect the master disk if it does not already come that way. Some operating systems provide a software write-protect feature, but use physical protection too for safety.

Some CP/M-like implementations provide a disk copy command, named something like COPY, which accomplishes the above 3 steps using a single command. However, COPY may be suitable only for systems having more than one drive.

If there is no appropriate COPY command for your system, then separate commands like the following are probably necessary:

1. Some CP/M-like implementations provide a formating command named probably FORMAT or INIT. If not, don't panic: Disks generally come preformatted from the manufacturer anyway.

2. Most CP/M-like implementations provide a command or pair of commands for generating on a formatted disk a version of the operating system sized to take full advantage of the current amount of memory. For example, the pair of commands may be

```
MOVCPM * *
SYSGEN
```

3. For systems having more than one drive, the transient CP/M **PIP** (**P**eripheral **I**nterchange **P**rogram) provides a means of copying files from one disk to another. For this purpose, the appropriate form of the command is

```
PIP destination=source
```

As a convenience, CP/M generally permits an asterisk, *, to be used in place of a name in order to indicate all possible names in that position. Thus, most CP/M-like systems permit the command

```
PIP A:=B:*.*
```

which transfers all of the files to drive A from drive B. Since PIP involves writing information on a disk, heed the warning in Section 4.6.3 concerning the need for a warm-start after inserting a new

disk.

4. The transient CP/M **DDT** (**D**ynamic **D**ebugging **T**ool) command and intrinsic CP/M **SAVE** command together provide a way of copying files between disks even on single-drive systems as follows:

   a. Issue the command

      DDT

   b. In response to the DDT prompt "*", enter a DDT command of the form

      Ifiletobecopied

   c. Insert the disk containing the file to be copied, then enter the DDT command

      R

   d. Insert the disk that is to receive the copy then type CTRL-C to exit from DDT back to the operating system.

   e. Enter a CP/M command of the form

      SAVE #records newcopydesignation

      where "#records" is the number of 256-byte records to be transferred from memory to the disk. The way to determine this number is to take the first two digits of the 4-digit hexadecimal number displayed as the value of NEXT after completion of the DDT R command, and convert it to decimal for use in SAVE. (To convert, multiply the first digit by 16 then add the second digit, noting that A means 10, B means 11, C means 12, D means 13, E means 14, and F means 15.)

Remarks:

1. For some implementations, the FORMAT, MOVCPM, or SYSGEN commands may be awkward or impossible to use on a single-drive system without resorting to use of a built-in ROM monitor or front-panel switches.

2. Due to wear and inadequate industry-wide manufacturing standards, there are slight or not-so-slight mechanical, electrical and magnetic differences between various nominally compatible drives and disks. Consequently, we suggest that if you have two or more drives, first try placing the new disk in the drive on which it will be used most often. The master disk can then be tried on each of the other drives, trading with the new disk if none of the other drives are successful.

3. For some operating systems it may be necessary to remove write protection from the old disk temporarily, even though it is only to be read. (Perhaps this is true only if the old disk is in the primary drive.) Moreover, it may be awkward to copy a disk without first temporarily removing the write protection long enough to copy the operating system onto that disk, especially if there is only one

drive. Naturally, the supplied master disk does not ordinarily come with a copy of your operating system on it, because the distributors of muSIMP/muMATH do not wish to violate the copyright on your operating system. Also, installed versions of operating systems generally depend on many machine-dependent parameters including memory size together with the specific type of terminal, disk, printer, etc.

4. Become thoroughly familiar with the terminal, computer, disk drives, and operating system. Most cases of accidental overwriting of the master disk are committed in the first few moments by eager users who are inexperienced with the system on which they are installing the new software. In particular, practice initializing a disk then generating a disk operating system on it. Practice transfering files from a spare, write-protected disk to the new disk.

5. If you cannot read the master disks after several attempts and after carefully restudying our directions and those provided for the hardware and operating system, then:

   a) Carefully check to see whether you correctly specified all of the details on the order form and purchased the proper type of blank disks.

   b) Use a head-cleaning and an alignment test disk or have your drive checked professionally.

   c) Contact the dealer or distributor who sold you the disk.

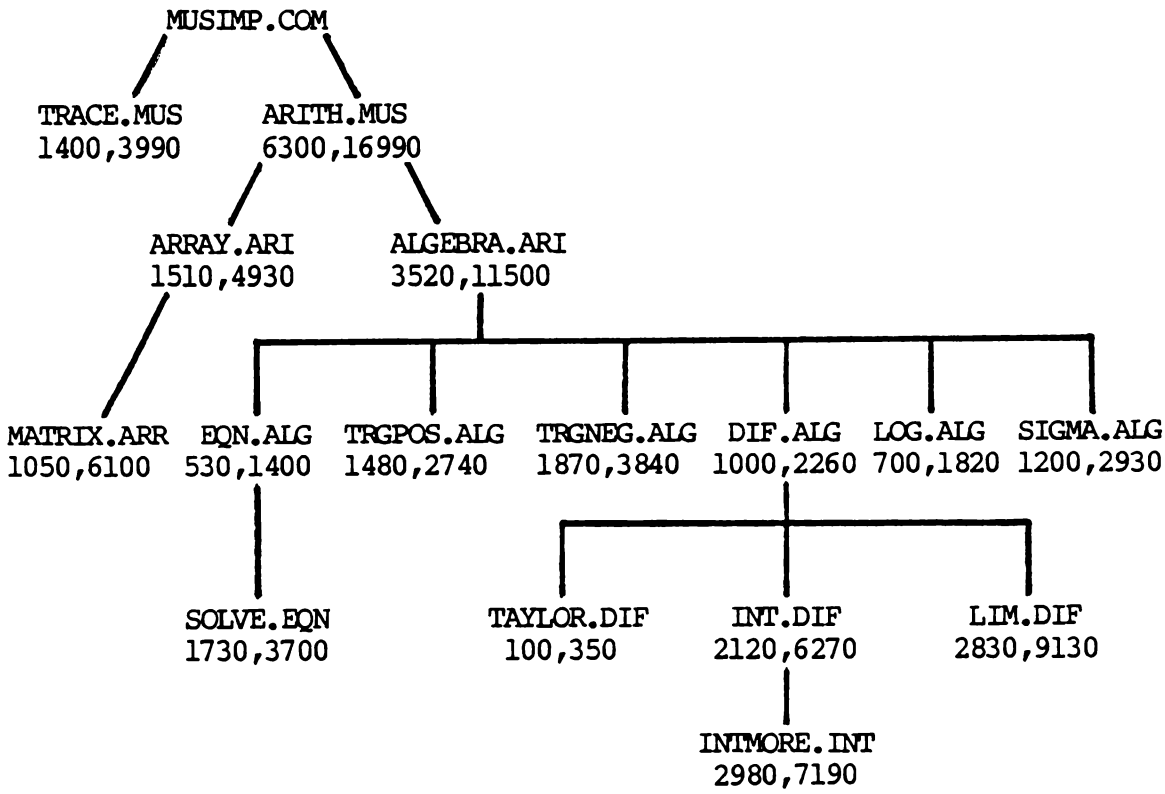
5. FILE HIERARCHY, SIZES & NAMING CONVENTION

The **muSIMP incremental compiler-interpreter** is distributed as a directly executable machine language COMmand file designated MUSIMP.COM. The only other COM file on the master disk is a general-purpose job-continuation utility designated CONTINUE.COM.

The **muMATH-80 Symbolic Math System** is distributed as a set of source files written in muSIMP. Each source file provides a package of mathematical capabilities. The packages are related in a modular fashion so as to accommodate differing mathematical needs and differing computer memory sizes. In the Table of Contents to this manual, the titles of subsections 9.1, 9.2, ... briefly indicate the purpose of each such package. The corresponding subsections explain how to use these packages.

Besides the above mentioned files, the master disk(s) contains a set of calculator mode and a set of programming mode lesson files. Their respective subjects are briefly described in the Table of Contents under sections 15 and 16.

The more sophisticated mathematical packages require prerequisite files as indicated by the following **dependency diagram:**



Each muMATH package given in the above diagram requires as mandatory prerequisites all those packages between it and the top of the tree (i.e. beginning with MUSIMP.COM). For example, ALGEBRA.ARI requires ARITH.MUS as a prerequisite, which in turn requires MUSIMP.COM.

muMATH systems are constructed by loading MUSIMP.COM and then working down the tree until all the desired packages are loaded. Depending on the application, files other than the mandatory prequisites should also be loaded because they can provide potentially helpful simplifications. For instance, it is advisable to load LOG.ALG when using EQN.ALG or INT.DIF on problems involving logarithms. When using INTMORE.INT to evaluate improper integrals or when using SIGMA.ALG to evaluate infinite series it is necessary to load LIM.DIF.

The first number below a package's name is the approximate number of bytes required by the package when loaded into muSIMP. As described in Section 6.4, this number can be used to determine whether your computer system has sufficient memory capacity to store a given set of packages. The second number of each pair is the number of bytes in the package's source file on the disk.

All of the files provided on the master diskette(s) or generated by the muSIMP SAVE command are named in accordance with the following conventions:

1. **COM** files are unprintable machine language programs which are directly executable as operating system commands.

2. **SYS** files are unprintable memory-images generated by the muSIMP SAVE command and executed by the LOAD command.

3. **CLES** files are interactive calculator mode lessons which can be executed from within muSIMP by the RDS command. The proper order of the lessons is indicated by the numeric suffix **n**.

4. **PLES** files are interactive programming mode lessons which can be executed from within muSIMP by the RDS command. The proper order of the lessons is indicated by the numeric suffix **n**.

5. All files on the master diskette(s) of a type other than COM and SYS are muSIMP program source files. The type of these files denotes the first three letters of their immediate prerequisite file. For example:

   a) File **ALGEBRA.ARI** contains the algebra package and requires the file **ARITH.MUS** as a prerequisite.

   b) File **CLES3.ALG** is an interactive calculator mode lesson file and requires the file **ALGEBRA.ARI** as a prerequisite.

muSIMP source files can be generated by any **text editor** that can produce standard CP/M text files without line numbers. A suitable example of such an editor is the standard ED program provided with CP/M.

6. HOW TO START muSIMP PROGRAMS SUCH AS muMATH

Initiating muMATH is easy once a memory image **SYS file** has been constructed which includes all the muMATH packages that you wish to use. Section 6.4 describes how to construct such a system, and Section 7 describes how to save the memory image as a SYS file.

The name given to a SYS file is normally that of the highest-level math package which it contains, as indicated by the dependency diagram of Section 5. For example, **ALGEBRA.SYS** might be the designation of a SYS file containing ARITH.MUS and ALGEBRA.ARI. However, if the SYS file includes packages from more than one branch of the dependency diagram, then the name should be a combination of both packages. For example, the SYS file containing both ARRAY.ARI and SOLVE.ALG as well as their prerequisites might be named **ARRSOL.SYS.**

The CP/M limitation of 8-character file names effectively limits the name to combinations of only 2 or 3 package names. Consequently, more generic names such as CALCULUS.SYS or even MATH.SYS are good choices in such cases. An adhesive label on the diskette should be used to identify the highest-level components of such SYS files.

6.1 USING MEMORY-IMAGE SYS FILES

To start execution of muSIMP with a SYS file, one merely enters an operating-system level command of the form

drv1:MUSIMP drv2:nam

Here drv1 represents the drive that MUSIMP.COM is on, and drv2:nam represents the drive and name of the desired SYS file. To start execution of muSIMP alone, one merely omits the SYS file argument, issuing a command of the form

drv1:MUSIMP

As always in CP/M commands, the drive prefixes can be omitted from a file's name if the file is on the currently-logged (i.e. default) drive. For example, if MUSIMP.COM is on the default drive and you wish to load ALGEBRA.SYS from drive C, issue the CP/M command

MUSIMP C:ALGEBRA

In a few seconds, muSIMP should display a **sign-on** message of the form:

muSIMP-80 2.xx (mm/dd/yy)
Copyright (C) 1981 MICROSOFT
Licensed from The SOFT WAREHOUSE

where appropriate numbers appear for the version, month, day, and year. This version number and date should be included in all inquiries

concerning muSIMP or muMATH.

After displaying the sign-on message, muSIMP proceeds to load the SYS file if one was given. The SYS file should take less than half a minute to load, depending on its size. Then, muSIMP issues a question mark prompt, as described in the next subsection.

6.2 THE INTERACTION CYCLE

The muSIMP **prompt** is a question mark followed by a space. Its presence indicates the system is ready to accept an expression entered from the terminal. The user then enters an expression followed by a semicolon and a RETURN. The expression is first **parsed** into an internal representation. Next, an "@:" is displayed on a new line to herald the "@nswer". After that the expression is **evaluated**, and then a space is printed to indicate that the evaluation phase is complete. Finally the result is deparsed and printed in mathematical notation. After 2 more blank lines are printed, the muSIMP prompt is displayed again. This interaction cycle is repeated until the **SYSTEM command** is issued by the user. The SYSTEM command exits muSIMP and returns control to the operating system.

For example, here is a segment of a trivial muSIMP dialogue:

```
? 2 + 5 - 3;

@: 4


? JOHN = MARY;

@: FALSE


? MEMBER (APPLE, '(GRAPE, APPLE, PLUM));

@: TRUE
```

These three examples show use of some of the arithmetic operators, the equality comparison operator, and a set membership function. The calculator-mode lessons described in section 8 introduce many of the powerful built-in operators and functions. Here we are concerned merely with the mechanics of interacting with muSIMP.

The line-editing features of the host operating system are operative: As described in Section 4, control characters such as CTRL-H, CTRL-X, CTRL-S and CTRL-P can be used to correct errors on the current line, suspend scrolling, and toggle output to the printer.

As soon as a RETURN is entered, muSIMP parses that line of the expression. However, evaluation of the expression does not occur until a semicolon terminating the expression has been read. If a line of the

expression contains a **syntax error** (i.e. a grammatically incorrect form), an error message occurs and the remainder of the expression following the error is displayed on the console. This should help to quickly identify the cause of the problem. When a syntax error does occur, a semicolon must be entered to terminate the expression if this has not already been done.

However, if the terminated expression is syntactically correct but not what was intended, muSIMP proceeds with evaluation. If the evaluation turns out to be quite time-consuming, you may wish to interrupt it as described in the next section. Otherwise, the simplest and safest thing to do is to let the evaluation proceed to completion, after which you can enter the expression that was desired.

If a terminating semicolon has not been entered following an undesired but syntactically correct expression on a previous line, one can purposely enter a syntax error followed by a semicolon in order to avoid the evaluation phase. For example, one could terminate the expression with:

```
(;
```

The value of the variable named **NEWLINE** controls the number of blank lines between the muSIMP input and answer lines: NEWLINE is initially 1, resulting in 1 blank line following each input and 2 blank lines following each output. To reduce these to 0 and 1 respectively, allowing more to fit on a CRT screen at the expense of reduced readability, enter the muSIMP command

```
NEWLINE: 0;
```

This assignment sets the value of NEWLINE to 0. Assignments are more thoroughly explained in lesson CLES1.ARI.

6.3 INTERRUPTING EVALUATION

An evaluation in progress can be aborted by depressing either the **ESC**ape or **ALT**mode key. For terminals with neither of these keys, typing a CTRL-[(i.e. typing a left bracket while holding down the control key) is equivalent to ESCape. The following is the **interrupt option menu:**

```
*** INTERRUPT:  To Continue Type: RET;
Executive: ESC, ALT;  System: CTRL-C?
```

1. The usual choice is typing another **ESC**ape or **ALT**mode, which starts a new interaction cycle. All assignments, function definitions, and property values existing at the instant of interruption are preserved. Since many muMATH operations temporarily alter control variables during computation, it is advisable to issue the muMATH command **FLAGS()** after such an interruption in order to determine if any control variables need to be restored to desired values.

2. The **RET**urn choice is appropriate if one wishes to let the evaluation proceed after all.

3. The **CTRL-C** choice is appropriate if one wishes to exit muSIMP, and return control to the operating system. This would be the reasonable response if the muSIMP environment has become hopelessly cluttered and it is desired to start a new session from scratch.

Naturally, it is always possible to interrupt muSIMP by RESETing the computer as described in Section 4.5. A RESET done in the middle of an evaluation leaves the interrupted muSIMP **process** in an unpredictable state that generally can not be correctly resumed. Thus, using the machine RESET to interrupt an evaluation forces one to start a fresh muSIMP process.

6.4 READING muSIMP SOURCE FILES

If a SYS file incorporating the desired packages does not exist, then the desired source files can be read into memory using a **ReaD Select command** of the form

RDS (nam, typ, drv);

where "nam" stands for the file name, "typ" stands for its type, and "drv" stands for the drive on which it is mounted. The drive is an optional argument that defaults to the drive currently logged.

> Note that the file name and type are separated by a comma rather than a period, and that the drive requires only a single letter without the customary colon used at the operating system level.

When loading a series of packages, the lower-level packages (i.e the ones nearer the top of the dependency diagram in Section 5) should be loaded before the higher-level ones. One can start from either muSIMP alone or, to save time, from muSIMP together with a SYS file incorporating a subset of the desired packages.

Each muMATH source file requires computer memory space to store its function definitions, etc., so it is important to be aware of the required space and the remaining available space when using RDS. Accordingly, it is important to know that:

1. The first number under each package name in the dependency diagram of section 5 is the approximate number of bytes occupied by the package when read into muSIMP.

2. The number of remaining available **bytes** of storage is returned by the muSIMP command

RECLAIM ();

3. muSIMP requires some of the available space for a **control stack** together with numeric or symbolic data assigned to variables or generated during intermediate calculations. Leaving insufficient space for these purposes will provoke the "space exhausted" error

message, or at least seriously slow the evaluation process. The required amount of such space depends upon the type of expressions that are being evaluated. Some judgement about this will be gained with experience using muSIMP. The recommended absolute minimum available space is about 4K (i.e. 4000) bytes, and more is better.

As an example of the use of RDS in conjunction with RECLAIM to cautiously build a particular combination of packages, suppose that we wish to build an environment containing the equation package and its prerequisites, beginning from a SYS file incorporating ARITH.MUS. We could proceed as follows:

```
A>MUSIMP ARITH

muSIMP-80 2.10 (04/11/81)
Copyright (C) 1981 MICROSOFT
Licensed from The SOFT WAREHOUSE

? RECLAIM ();
@: 30000

? RDS (ALGEBRA, ARI);
@: ALGEBRA

? RECLAIM ();
@: 26500

? RDS (EQN, ALG);
@: EQN
```

The space remaining immediately after initiating muSIMP depends upon the amount of contiguous RAM memory space below the permanent portion of the operating system, which varies widely with the hardware and the particular operating-system variant. Consequently, the above sequence is likely to yield different numbers on most computers. Even the space required by a package varies somewhat depending on the environment into which it is read, because muSIMP optimizes by sharing common subexpressions between each function definition and all previous ones.

muSIMP permits more than one expression per line, so when certain that a particular combination of packages will fit, it is convenient to put several RDS commands on a line. For example:

```
? RDS(ALGEBRA,ARI); RDS(EQN,ALG); RDS(SOLVE,EQN);
```

Source files are simply a sequence of muSIMP commands that could be entered just as well from the console if you wanted to retype them every time. As these commands are read in, muSIMP parses them into an internal form, then evaluates them. When the command is a function definition, the definition is **pseudo-compiled** into an extremely compact form. This requires searching all previously compiled functions for any subexpressions that can be shared with the new function.

Although remarkably fast for such exhaustive space optimization, the compilation and parsing of such source files require considerably more time than loading an equivalent memory-image SYS file. For example, the muSIMP source listing for ARITH.MUS and ALGEBRA.ARI, the two largest muMATH packages, is about 20 printed pages. To read these files takes approximately 7 minutes total using the RDS command on a microcomputer running at 2 MHz (megahertz), versus about 1/2 minute to load the corresponding SYS file. This speed differential is the major reason for generating SYS files.

As described in Section 7, building SYS files is a relatively simple matter. Nevertheless, we recommend not bothering with the generation of SYS files until after taking the first few muMATH lessons. Until then, just RDS the necessary files for each session.

7. SAVING AN ENVIRONMENT MEMORY IMAGE

The muSIMP **SAVE command** saves a compact memory image of the static environment existing at the time the command is executed. A muSIMP **environment** consists of all the variable values, function definitions and property values which have been entered directly from the terminal or read from a source file since muSIMP was started or since the most recently executed **LOAD command** was issued. Reasons for wanting to save an environment include the following:

1. to speed and simplify the startup process for commonly-used combinations of muMATH packages;

2. to save the state arrived at by an interactive dialogue for continuation at a later time;

3. to **checkpoint** the state arrived at by an interactive dialogue so that the exact same state is easy to restore after exploring each of several alternative continuations, some of which might alter the environment in a way that is awkward or tedious to restore;

4. to save a "package" developed by interactively entering muSIMP function definitions, properties and assignments.

7.1 THE muSIMP SAVE COMMAND

The muSIMP SAVE command has the general form

SAVE (nam, drv);

This command saves a **SYS file** of the indicated name on the indicated drive. As usual, the drive is optional, defaulting to the currently logged drive. The size of the resulting SYS file increases with the amount of information in the environment. As a rough guide, if **i** is the value returned by the muSIMP RECLAIM command when only muSIMP is present and **s** is the value returned by the RECLAIM command just before saving, then the size of the SYS file in kilobytes is approximately,

6 + (i-s)/1000

A diskette with sufficient free storage space to store the resulting SYS file must be used for the SAVE command. The amount of storage remaining on a disk can be determined by using the CP/M STAT command as described in section 4.6.2. STAT is a transient CP/M command which overlays and hence destroys the muSIMP program area. Hence determination of a diskette's free storage must be accomplished **before** beginning a muSIMP session. If there is insufficient disk storage space, muSIMP displays the following warning message on the console:

No Disk Space

If this message occurs, a fresh, blank diskette can be placed in the drive and the SAVE command attempted again.

7.2 OVERCOMING LOW-CAPACITY SINGLE-DRIVE SPACE LIMITATIONS

While operating under the CP/M type of operating system, diskettes on a given drive should not be changed at arbitrary times. The preferred time for changing diskettes is when your computer is at the CP/M command level, as indicated by the CP/M prompt. As described in section 4.6.3, a warm-start should be performed immediately after inserting a new diskette in a drive.

For computer systems with disk drives of relatively small storage capacity, it will be impossible to save a muMATH system on the disk which contained the source files required for the build. For single drive systems this means a fresh diskette must be placed on the drive to receive the SYS file produced by the SAVE command without being able to perform the warm-start procedure described above.

This **diskette change** is permissible provided the following cautions are observed: the drive is unloaded (i.e. drive motor inactive) and no disk file is currently being read from or written to on the diskette which is being replaced. In general, while running under muSIMP, diskettes can be changed as desired because muSIMP does a system reset whenever it is creating a new file on a disk.

For the more conservative in the crowd, an alternative method for changing diskettes is to temporarily exit the muSIMP/muMATH system using the muSIMP command:

```
SYSTEM ();
```

which returns control to the CP/M command level. At that point the diskette can be changed and a warm-start performed in the normal fashion by typing a CTRL-C. Finally the muSIMP/muMATH environment present at the time of exit can be restored by using the CP/M command:

```
CONTINUE
```

The **CONTINUE command** has the additional advantage of making it possible to check a diskette's DIRectory and perform any of the other **intrinsic** CP/M commands (see section 4.6.1). Also it **usually** permits return to a muSIMP/muMATH session inadvertently exited by typing a CTRL-C at the wrong time (see note following condition 2 below). Naturally, the use of any of the **transient** CP/M commands such as STAT, PIP, ED, or DDT will ruin the muSIMP environment making return impossible.

If the file CONTINUE.COM resides on drive drv, the general form of the CONTINUE command following the CP/M command prompt is

```
drv:CONTINUE
```

Since CONTINUE.COM is a zero length file, it merely results in a jump to the standard starting location for CP/M application programs. The following conditions are necessary for the utility to succeed:

1. The program which is being resumed must be designed for re-entry beginning at location 100H. It must store a flag that it can test to know whether it is being entered for the first time or merely being resumed.

2. The overlaying of high-end memory by a warm start must not damage the program or its data. During a warm start, the portion of CP/M (about 2K bytes) that handles transient commands is read into the high end of memory just below the "permanent" portion of CP/M.

Note: When the muSIMP SYSTEM command is issued, all data spaces are moved into low memory prior to exiting. Thus condition 2 above is satisfied, permitting restart of the muSIMP system. However, if exit results from typing an inadvertent CTRL-C **and** the muSIMP stack is less than the size of the memory which is overlayed by the CP/M transient command processor, it will be impossible to restore the muSIMP environment. For this reason we strongly encourage using the SYSTEM command to exit muSIMP rather than getting in the habit of typing a CTRL-C.

8. EXECUTING THE INTERACTIVE LESSONS

This section explains how to use the interactive muSIMP and muMATH lesson files. muMATH and the lessons are intended to serve a broad range of math levels from arithmetic through calculus, and a broad range of programming backgrounds from none to professional programmer. How is this scope possible? Read on:

The sequence of calculator mode lesson files named **CLES1, CLES2,** etc. explains how to use muMATH as an arithmetic or symbolic Calculator. The sequence of programming mode lessons named **PLES1, PLES2,** etc. explains how to write Programs in muSIMP, in order to enhance the suite of built-in operations or for any other purpose.

The **calculator sequence** is ordered according to the most common sequence in which the corresponding math subjects are taught. It is intended that a user proceed in this sequence only as far as his/her math background, before beginning the programming sequence. Due to slight variations in math curricula sequences, some users may prefer to skip certain calculator lessons in the middle of the sequence as well as at the end.

muMATH has such a rich set of built-in capabilities that many users will be content to postpone study of the programming sequence indefinitely. However, some users eventually will want to proceed to this sequence of lessons, perhaps for one or more of the following reasons:

1. to enhance the built-in muMATH capabilities,

2. to understand how the underlying muMATH algorithms work,

3. to learn computer programming,

4. to use muSIMP for some other application.

In order to make the **programming sequence** most useful to users of all mathematical backgrounds, the sequence begins with muSIMP examples that are non-mathematical or only arithmetic. Most general programming techniques and their realization in muSIMP are independent of higher-level math. Thus, only the last lesson in this sequence deals with muMATH specifically, explaining how to extend it, alter it, and even replace it with alternative symbolic math systems.

There are three ways to experience either set of lessons.

1. The best way is to execute them interactively, trying out examples at the opportunities provided.

2. The second best way is to read the printed record of a dialogue produced by someone else executing the lessons.

3. The third way is to read the listings of the lesson files printed in Sections 15 and 16 of this manual. However, this method provides only one side of an intended dialogue.

As indicated in Section 5, the first lesson you should take is CLES1.ARI. To commence this lesson you must first start-up a muMATH system containing at least the arithmetic package, ARITH.MUS. How to accomplish this is documented in Sections 4 to 6. Then, to actually start the lesson, simply issue the muSIMP command

```
RDS (CLES1, ARI, drv);
```

where "drv" is the name of the drive on which the disk containing CLES1.ARI is mounted. The lesson will then display informative text on your terminal telling you how to proceed from there.

If you become confused while taking a lesson, it may be helpful to read the printed listing of the lesson given in Section 15 or 16 of this manual. Also, taking the lessons with a companion is often worthwhile.

In any case, have fun!

9. USAGE OF INDIVIDUAL muMATH PACKAGES

This section describes in detail each package distributed with the muMATH-80 Symbolic Math System. Section 9.1 describes a general-purpose trace package useful when debugging muSIMP programs. The remaining subsections describe specific muMATH packages, presented in order of increasing mathematical sophistication.

9.1 TRACE.MUS: TRACE FACILITY

Purpose: File TRACE.MUS provides a trace package to help debug programs. Those using debugged muMATH packages in calculator mode should have no need for these facilities.

Prerequisite File: MUSIMP.COM

Control Variables:

1. **MATHTRACE** determines the notation (i.e. either mathematical or list) to be used when printing the trace. The default value is TRUE.

Usage:

```
TRACE (name1, name2, ...),
UNTRACE (name1, name2, ...).
```

Example:

```
? FUNCTION MEMB (EX1, EX2),                % A user defined function %
    WHEN EMPTY (EX2), FALSE EXIT
    WHEN EX1 = FIRST (EX2), TRUE EXIT
    MEMB (EX1, REST (EX2))
  ENDFUN$

? TRACE (MEMB)$                            % The TRACE command %

? MEMB ('DOG, '(CAT, COW, DOG, PIG))$      % A function application %

MEMB [DOG, (CAT, COW, DOG, PIG)]           % Computer generated output %
  MEMB [DOG, (COW, DOG, PIG)]
    MEMB [DOG, (DOG, PIG)]
    MEMB = TRUE
  MEMB = TRUE
MEMB = TRUE

? UNTRACE (MEMB)$                          % The UNTRACE command %
```

Remarks:

1. The trace of a function during the execution of a program provides an invaluable debugging tool.

2. Just before each traced function is applied to its evaluated arguments, the function name is displayed followed by the evaluated arguments enclosed in square brackets.

3. Just after return from each traced function, the function name followed by an equal sign and the returned value is displayed.

4. Indention is used to pair corresponding function calls and returns.

5. Functions are restored to normal by using the UNTRACE function.

6. If the control variable **MATHTRACE** is non-FALSE, the results of the trace are displayed in mathematical notation. If it is FALSE, the results are displayed in list notation.

9.2 ARITH.MUS: RATIONAL ARITHMETIC

Purpose: File ARITH.MUS can perform exact rational arithmetic operations including sums, differences, products, quotients, and powers, yielding exact results for numbers as large as 600 decimal digits. Numbers can be expressed in any desired radix base from 2 through 36 for both input and output. File ARITH.MUS also establishes much of the support necessary for algebraic transformations provided by file ALGEBRA.ARI.

Prerequisite File: MUSIMP.COM

Examples:

```
? 5/9 + 7/12;                 % Exact rational arithmetic %
@: 41/36

? FOO: (236 - 3*127) * -13;   % Make assignments to variables %
@: 1885

? FOO ^ 16;                   % Raise numbers to integer powers %
@: 254086547815589282275252071398862670233399963378906 25

? RADIX (2);                  % The radix base can be set from 2 %
@: 1010                       % through 36 %

? FOO;                        % Convert numbers between radix bases %
@: 11101011101

? 1011000101 + 111010001;     % Do binary arithmetic %
@: 10010010110

? RADIX (1010);               % Return to base 10 %
@: 2

? GCD (861, 1827);            % Compute the GCD of two numbers %
@: 21
```

Control Variables:

1. **PBRCH** is a control variable which, when nonFALSE, permits selection of a branch of a multiply-branched function. For ARITH.MUS, when PBRCH is nonFALSE the following simplification is performed:

   (expr1 ^ expr2) ^ expr3 —> expr1 ^ (expr2 * expr3)

   even when expr3 is not an integer. PBRCH is initially TRUE.

2. **ZEROBASE** is a control variable that, when TRUE, permits the simplification 0 ^ expr —> 1 even when expr is nonnumeric. ZEROBASE is initially FALSE because the transformation is invalid if expr represents a negative value.

3. **ZEROEXPT** is a control variable that, when TRUE, permits the simplification expr ^ 0 —> 1 even when expr is nonnumeric. ZEROEXPT is initially TRUE because the transformation is valid except when expr is identically zero.

4. **TRGEXPD** is a control variable that is initially 0. When TRGEXPD is a positive multiple of 7, then the complex exponential #E^#I is transformed to COS(1) + #I*SIN(1).

5. **LOGEXPD** is initially 0. When LOGEXPD is a positive multiple of 7, then all exponentials are converted to the base that is the value of variable LOGBAS.

6. **LOGBAS** is initially #E, which denotes the base of the natural logarithms.

Built-in Functions:

ABS (expr) is a function which returns the absolute value of its argument when the argument is a rational number. Otherwise, the rule-application function SIMPU is invoked, so the unevaluated absolute-value form is returned if no applicable rules are present.

ARGEX (expr) is a helper function used by SIMPU and elsewhere to appropriately partition an expression for application of a rule.

ARGLIST (expr) is a helper function used to appropriately group the operands of an expression for application of rules to operators such as "+" and "*" which take an arbitrary number of arguments.

BASE (expr) is a selector function that returns the base of an expression of the form base ^ exp; otherwise it returns expr itself.

CODIV (expr) is a selector function which returns the codivisor (meaning the non-numeric factors) of an expression that is a product. For nonproduct arguments, CODIV returns 1 if the argument is numeric, returning the entire argument otherwise.

COEFF (expr) is a selector function which returns the coefficient (meaning the numeric factors) of an expression that is a product. For nonproduct arguments, COEFF returns 1 if the argument is not a number, returning the entire argument otherwise. In all cases

expr = COEFF (expr) * CODIV (expr)

DEN (expr) is a selector function that returns the denominator of its argument, returning 1 when there is none.

DENOM (expr) is a recognizer function that returns TRUE iff its argument has the internal form (^ bas exp), with exp being negative or having a negative coefficient.

EVSUB (expr, subexpr, replacement) is a function that returns the result of evaluating a copy of its first argument, wherein each syntactic occurrence of its second argument is replaced by the third argument.

EXPON (expr) is a selector function that returns the exponent of an expression of the form base ^ exp; otherwise it returns 1. Note that in all cases

expr = BASE (expr) ^ EXPON (expr)

GCD (intgr1, intgr2) is a function that returns the positive greatest common divisor of its integer arguments.

LCM (intgr1, intgr2) is a function that returns the positive least common multiple of its integer arguments.

MIN (intgr1, intgr2) is a function that returns the minimum of its two integer arguments.

MULTIPLE (intgr1, intgr2) is a function that returns FALSE if its second integer argument is not an integer multiple of its first integer argument.

NEGCOEFF (expr) is a recognizer function that returns TRUE iff its argument is negative or has a negative coefficient, returning FALSE otherwise.

NEGMULT (intgr1, intgr2) is a predicate that returns TRUE iff its second integer argument is a negative integer multiple of its first integer argument.

NUM (expr) is a selector function that returns the numerator of its argument, returning the entire argument when there is no denominator.

NUMBER (expr) is a recognizer function that returns TRUE iff its argument is an integer or a rational number.

POSMULT (intgr1, intgr2) is a predicate that returns TRUE iff its first integer argument is a positive multiple of its second integer argument.

POWER (expr) is a recognizer function that returns TRUE iff its argument is of the form expr1 ^ expr2, returning FALSE otherwise.

PRODUCT (expr) is a recognizer function that returns TRUE iff its argument is of the form expr1 * expr2, returning FALSE otherwise. It is important to realize that quotients are represented as products involving negative powers.

RECIP (expr) is a recognizer function that returns TRUE iff its argument is a rational number of the form 1/d, returning FALSE otherwise.

SIMPU (name, expr) is a function that applies any appropriate established rules for the unary function or operator whose name is the first argument of SIMPU and whose operand is the second argument of SIMPU.

SUB (expr, subexpr, replacement) returns a copy of its first argument, wherein every syntactic instance of its second argument is replaced by its third argument. In general this produces an unsimplified result, so the similar EVSUB function uses SUB, then EVAL.

SUM (expr) is a recognizer function that returns TRUE iff its argument is of the form expr1 + expr2, returning FALSE otherwise. It is important to realize that differences are represented as sums involving terms having negative coefficients.

Optional Fractional Power Sub-package:

Purpose: This Portion of ARITH.MUS provides the facilities for the simplification of fractional powers of numbers and complex exponentials. Comments in file ARITH.MUS identify the boundaries of this portion, which can be deleted with a text editor if space is scarce and there is no need for this portion.

Usage:

number ^ (fraction),
#E ^ (intgr * #I * #PI / 2).

Examples:

```
? (-24) ^ (1/3);
@: -2 * 3 ^ (1/3)

? (-4) ^ (1/2);
@: #I * 2

? #E ^ (3 * #I * #PI / 2);
@: - #I
```

Control Variables:

1. **PBRCH**, if FALSE, prevents Picking a BRanCH for fractional powers. For example, 4 ^ (1/2) will not simplify to 2 if PBRCH is FALSE. However, PBRCH is initially and normally TRUE.

Remarks:

1. The following variables are used throughout muMATH:
   a. **#E** denotes the base of the natural logarithms (i.e. 2.71828...);
   b. **#I** denotes the positive square root of minus one (-1)^(1/2);
   c. **#PI** denotes the ratio of the circumference of a circle to its diameter (i.e. 3.1415...).

2. Simplification of fractional powers takes place only if the control variable **PBRCH** is non-FALSE. The positive real branch is selected if one exists. Otherwise, the negative real branch is selected if one exists. Otherwise, the branch with the smallest positive argument is selected.

3. As in manual computations, Picking a BRanCH of a fractional power involves an arbitrary choice that can yield invalid results. Thus, the user is cautioned to verify results obtained by such operations.

4. The global variable **PRIMES** contains a list of successive primes, beginning with the integer 2. For fractional powers, the radicand is factored into a product of powers of the numbers in PRIMES, perhaps times a residual having no factors in PRIMES. The fractional power is then distributed over this product, with a discrete variant of Newton's method being used to determine if the fractional power of any residual is an integer. Thus, simplification of fractional powers of large integers might be incomplete if the list PRIMES does not contain a sufficient number of primes.

5. As in manual computations, reduction of complex exponentials modulo (2 * #PI * #I) is inconsistent with the identity

   LN(Z*W) = LN(Z) + LN(W)

   Thus, the user is cautioned to verify results obtained using both transformations together.

Optional Factorial Package:

Purpose: This portion of ARITH.MUS provides the factorial postfix operator "!". The factorial of a non-negative integer, N, is recursively defined as follows:

```
N! = 1,          if N = 0;
N! = N*(N-1)!,   for N > 0.
```

Usage:

```
N!
```

Example:

```
? 5!;
@: 120
```

Remarks:

1. The left binding power of "!" is 160, which is very high. Thus -5! parses to -(5!) and 3^5! parses to 3^(5!).

2. When not given a nonnegative integer operand, "!" calls upon the SIMPU rule-application function, thus returning the unevaluated factorial form if no appropriate rules are established.

9.3 ALGEBRA.ARI: ELEMENTARY ALGEBRA

Purpose: File ALGEBRA.ARI provides the basic algebraic simplification and transformations required for expressions using the elementary operators "+", "-", "*", "/", and "^". Simplifications are usually automatic whereas transformations must be explicitly requested by means of CONTROL VARIABLES.

Prerequisite File: ARITH.MUS

Examples:

```
? 5*X^2/X - 3*X^1;
@: 2 * X

? (5*X)^3 / X;
@: 125 * X^2

? EXPD ((3*Y^2 - 2*Y + 5)^3);
@: 125 - 150*Y + 285*Y^2 - 188*Y^3 + 171*Y^4 - 54*Y^5 + 27*Y^6

? FCTR (6*X^2*Y - 4*X*Y^2/Z);
@: 2*Y*X*(-2*Y+3*X*Z) / Z
```

Automatic Simplification:

1. Rational arithmetic is used to combine numerical operands. (See Section 9.2 for a complete description.)

2. Identities and zeros are appropriately applied to expressions. For example:
   `0+X --> X;  1*Y --> Y;  0*Z --> 0;`

3. Sums and products are flattened and uniquely ordered to facilitate expression comparisons. For example:
   `X+(Y+Z) --> X+Y+Z;  Z*(Y*X) --> X*Y*Z;`

4. Similar terms and products are combined. For example:
   `3*X + 2*X --> 5*X;  X^5 / X^2 --> X^3;`

5. Powers of #I (i.e. the square root of -1) are reduced. For example:
   `#I^7 --> -#I;`

Control Variables:

The control variables described in this section enable the muMATH user to have complete control over the rules used to simplify an expression. However, they are rather difficult for the novice to master. Therefore the utility functions EXPAND, EXPD, and FCTR (described below) have been included in muMATH to make it easy to obtain the most common forms of an expression without the need to individually set control variables. We recommend that these functions be used until more precise control of the control variables is required.

1. **NUMNUM** controls the distribution (factoring) of factors in the NUMerator of an expression over (from) a sum in the NUMerator.

   Identity: $A * (B+C) \quad <--> \quad A*B + A*C$

2. **DENDEN** controls the distribution (factoring) of factors in the DENominator of an expression over (from) a sum in the DENominator.

   Identity: $$\frac{1}{A} * \frac{1}{B + C} \quad <--> \quad \frac{1}{A*B + A*C}$$

3. **DENNUM** controls the distribution (factoring) of factors in the DENominator of an expression over (from) a sum in the NUMerator.

   Identity: $$\frac{1}{A} * (B+C) \quad <--> \quad \frac{B}{A} + \frac{C}{A}$$

4. **NUMDEN** controls the distribution (factoring) of factors in the NUMerator of an expression over (from) a sum in the DENominator.

   Identity: $$A * \frac{1}{B + C} \quad <--> \quad \frac{1}{B/A + C/A}$$

5. **BASEXP** controls the distribution (factoring) of the BASe of an expression over (from) the EXPonent.

   Identity: $A^{(B+C)} \quad <--> \quad A^B * A^C$

6. **EXPBAS** controls the distribution (factoring) of the EXPonent of an expression over (from) the BASe.

   Identity: $(A*B)^C \quad <--> \quad A^C * B^C$

7. **PWREXPD** controls whether or not integer PoWeRs of sums are EXPanDed in numerators and/or denominators.

8. **ZEROEXPT** controls the use of the following identity that is valid for all A not equal to 0.

   Identity: $A^0 \quad --> \quad 1$

9. **ZEROBASE** controls the use of the following identity which is only valid for positive A.

   Identity: $0^A \quad --> \quad 0$

Remarks:

1. For the first six of the above control variables, the kinds of factors, bases, or exponents that are distributed or factored from the expression can be precisely controlled by assigning appropriate values to the respective control variable. Positive integer values cause distribution whereas negative values cause

factoring. The exact type of expression which is distributed or factored can be determined from the following table:

Prime	Type	Examples
2	Numerical expressions	4, -1/3, 5/7
3	Other non-sums	X, SIN(Y), Z^3
5	Sums	R+S, X^2-X, LN(X)+Z

Therefore, if a control variable is a multiple of one or more of the above primes, then that type of expression is distributed or factored in accordance with that control variable's identity transform.

2. The following is an example of the use of NUMNUM to control the distribution of factors over a sum. Note that differences are internally represented as sums involving negative coefficients.

```
3 * X * (1+X) * (1-X)  -->
    3 * X * (1+X) * (1-X)          if NUMNUM is 0,
    X * (3+3*X) * (1-X)            if NUMNUM is 2,
    3 * (X+X^2) * (1-X)            if NUMNUM is 3,
    3 * X * (1-X^2)                if NUMNUM is 5,
    (3*X+3*X^2) * (1-X)            if NUMNUM is 6,
    X * (3-3*X^2)                  if NUMNUM is 10,
    3 * (X-X^3)                    if NUMNUM is 15,
    3*X - 3*X^3                    if NUMNUM is 30.
```

3. As another example, if DENDEN is 15, then

```
Y / 3 / X / (1+X) / (1-X)  -->  Y / (3*(X-X^3)).
```

4. As another example, if DENNUM is 6, then

```
(X+3) / 3 / X  -->  1/3 + 1/X.
```

5. When **PWREXPD** is a positive integer multiple of 2, then multinomial expansion occurs in numerators. When PWREXPD is a positive integer multiple of 3, then multinomial expansion occurs in denominators. Thus, for example, when PWREXPD is 6,

```
(1+X)^3 / (1+X+Y)^2  -->
          (1+3*X+3*X^2+X^3) / (1+2*X+2*Y+2*X*Y+X^2+Y^2).
```

6. The importance of becoming thoroughly familar with the use of PWREXPD, NUMNUM, DENDEN, and DENNUM cannot be over-emphasized! muMATH-80 cannot read a user's mind, so these control variables are the major means of specifying which of the many alternative transformations are desired at each stage in a dialog.

7. The remaining control variables are of less frequent concern, but changing their settings is occasionally crucial to achieving a desired effect. Since they follow the same general scheme, they are easy to use after the more important control variables have been mastered.

8. **NUMDEN** controls the distribution of a numerator over a denominator. Thus, this transformation yields a kind of "continued-fraction" expansion. For example,

```
(3+X) / (1+X)  -->
    1 / (3/(1+X) + X/(1+X))          if NUMDEN is 5,
    1 / (1/(1/3+X/3) + 1/(1/X+1))    if NUMDEN is 30.
```

9. **BASEXP** controls the distribution (factoring) of exponents over the base of expressions. It effect is illustrated as follows:

```
2 ^ (1+N)   -->  2 * 2^N
                 if BASEXP is a positive multiple of 2,

X ^ (1+N)   -->  X * X^N
                 if BASEXP is a positive multiple of 3,

(A+B) ^ (1+N)  -->  (A+B) * (A+B)^N
                 if BASEXP is a positive multiple of 5.
```

The opposite transformation (i.e. collecting common bases under an exponent) occurs when BASEXP is negative. Since that is usually more desirable, the default value for BASEXP is -30.

Built-in Functions:

FLAGS () prints the current value of the system control variables which are members of the list assigned to the variable **FLAGS.** The default of FLAGS is: (PWREXPD, BASEXP, EXPBAS, NUMDEN, DENDEN, DENNUM, NUNNUM, PBRCH, TRGEXPD, LOGEXPD, LOGBAS, ZEROBASE, ZEROEXPT).

FREE (expr, indet) is a predicate which returns true if and only if expr contains no occurrences of the variable indet.

EXPAND (expr) evaluates expr to yield a fully expanded denominator distributed over the terms of a fully expanded numerator. The following temporary assignments are made:

```
NUMNUM: DENDEN: DENNUM: EXPBAS: 30;
PWREXPD: 6;    BASEXP: -30;   NUMDEN: 0;
```

EXPD(expr) evaluates expr to yield a fully expanded numerator over a fully expanded denominator. The following temporary assignments are made:

```
NUMNUM: DENDEN: EXPBAS: 30;
PWREXPD: 6;  NUMDEN: 0;   DENNUM: BASEXP: -30;
```

FCTR (expr) evaluates expr to yield a semi-factored numerator over a semi-factored denominator. The following temporary assignments are made:

```
DENNUM: BASEXP: -30;   EXPBAS: 30
PWREXPD: NUMDEN: 0;  NUMNUM: DENDEN: -6;
```

Control-Variable Summary:

Control Var.	Initial Value	Positive Transformation	Negative Transformation
NUMNUM	6	A*(B+C) —> A*B + A*C	A*B + A*C —> A*(B+C)
DENDEN	6	$\frac{1}{A} * \frac{1}{B+C} \rightarrow \frac{1}{A*B + A*C}$	$\frac{1}{A*B + A*C} \rightarrow \frac{1}{A} * \frac{1}{B+C}$
DENNUM	6	$\frac{B+C}{A} \rightarrow \frac{B}{A} + \frac{C}{A}$	$\frac{B}{A} + \frac{C}{A} \rightarrow \frac{B+C}{A}$
NUMDEN	0	$\frac{A}{B+C} \rightarrow \frac{1}{B/A + C/A}$	$\frac{1}{B/A + C/A} \rightarrow \frac{A}{B+C}$
BASEXP	-30	A^(B+C) —> A^B*A^C	A^B*A^C —> A^(B+C)
EXPBAS	30	(A*B)^C —> A^C*B^C	A^C*B^C —> (A*B)^C
PWREXPD	0	(A+B)^N —> A^N+...+B^N	(A+B)^-N —> $\frac{1}{A^N+...+B^N}$

9.4 EQN.ALG: EQUATION SIMPLIFICATION

Purpose: File EQN.ALG provides a facility for treating equations as expressions that can be assigned to variables, added, multiplied, squared, etc. Thus equations can be solved manually with this package.

Prerequisite File: ALGEBRA.ARI
Also helpful is file LOG.ALG if an equation involves logarithms, or files TRGPOS.ALG and TRGNEG.ALG if an equation involves trigonometric functions.

Usage:
expr1 == expr2.

Examples:

```
? EQN:  5*X - 3*X - 7 == 2 + 4;
@: 2*X - 7 == 6

? EQN + (7 == 7);
@: 2*X == 13

? @/2;
@: X == 13/2

? [2*X == 6, 4*Y == 8] / 2;
@: [X == 3, 2*Y == 4]
```

Remarks:

1. The two sides of the equation are independently simplified according to the current control settings. However, no attempt is made to solve the equation by automatically shifting terms from one side of the equal sign to the other. Moreover, no attempt is made to verify that the equation is an identity or has a solution.

2. The use of == to indicate equations should not be confused with the use of = to indicate the equality predicate. The = operator is commonly used within the WHEN-EXIT construct in muSIMP function definitions. When used in this more active role, the result is always either TRUE or FALSE depending upon whether or not the left and right sides are equal.

3. The left and right binding powers of == are 80, which is the same as for =.

4. As illustrated by the above example, when a non-equation is combined with an equation, the non-equation is independently combined with both sides.

5. Although the above example illustrates how equations can be solved stepwise, file SOLVE.EQN automates this process.

6. Provided file ARRAY.ARI is loaded, sets of simultaneous equations can be represented as an array of equations. This is illustrated by the last example above.

7. As with manual computation, operations such as squaring both sides or clearing non-numeric denominators can enlarge the solution set, so the user should exercise caution and verify candidate solutions generated by such means.

8. If FOO is an equation, then SECOND (FOO) returns the left side of the equation, and THIRD (FOO) returns the right side.

9.5 SOLVE.EQN: EQUATION SOLVER

Purpose: File SOLVE.EQN provides a function for the exact solution of an algebraic equation.

Prerequisite File: EQN.ALG

Control Variables:

1. **PBRCH** determines whether or not SOLVE will pick a branch of a function's multiply branched inverse in order to solve an equation.

Usage:

SOLVE (equation, unknown).

Examples:

```
? SOLVE (X^2 = 4*A,  X);
@: {X = 2*A^(1/2)
    X = -2*A^(1/2)}

? SOLVE (LN(ATAN(X-1)) = B,  X);
@: {X = 1 + TAN(#E^B)}

? SOLVE (X^6 + 2*X^4 = -X^2, X);
@: {X = -#I,
    X = #I,
    X = 0}

? SOLVE (X^5 = -32, X);
@: {X = -2*#E^(8*#I*#PI/5),
    X = -2*#E^(6*#I*#PI/5),
    X = -2*#E^(4*#I*#PI/5),
    X = -2*#E^(2*#I*#PI/5),
    X = -2}
```

Remarks:

1. SOLVE returns a column vector of solutions. Column vectors are described in Section 9.6. The functions FIRST, SECOND, and THIRD can be used to extract individual solutions from a column of solutions. Alternatively, subscripts can be used for this purpose provided file ARRAY.ARI is loaded.

2. Forgetting the second argument of SOLVE is a frequent mistake.

3. As a convenience, when either side of an equation is zero, the = 0 can be omitted.

4. When no solution exists, SOLVE returns the empty column: {}.

5. When degenerate equations have an entire locus of solutions which require parameterization to represent completely, SOLVE introduces the arbitrary value parameters,

   ARB(1), ARB(2), ARB(3), ...

   The argument of ARB starts at 1 every time SOLVE.EQN or MATRIX.ARR is loaded. The following is an example of the solution to a degenerate equation:

   SOLVE (X == X, X); —> {X == ARB(1)}.

6. SOLVE expands the difference of two sides of an equation over a common denominator, then multiplies by the denominator to clear it. This multiplication can introduce spurious solutions if a zero of the denominator coincides with one of the numerator. Similarly, this multiplication can suppress a solution associated with an infinity of the denominator. Thus, the returned set should be regarded as candidates for some of the solutions rather than the complete verified solution set if an equation has a denominator that could be zero or infinite for finite values of the unknown. When these possibilities are present, it is the user's responsibility to verify the candidates by substitution or perhaps by using file LIM.DIF to take limits. It may be helpful in such instances to also use SOLVE to find any zeros of the common denominator in order to see if they coincide with any of those in the numerator.

7. After clearing the denominator, SOLVE attempts moderate factorization, then independently attempts to determine the zeros of each resulting factor. SOLVE recursively employs appropriate formulas for the inverses of the elementary functions and for the zeros of linear, quadratic, and binomial factors. When SOLVE encounters a factor that it cannot treat, it returns a "solution" of the form "factor == 0". Since the factor may be simpler than the original equation, it might serve as a useful point of departure for an approximate numerical solution.

8. If the control variable **PBRCH** is non-FALSE, SOLVE will pick a branch of a function's multiply branched inverse in order to solve an equation. Examples include such inverses as ATAN or fractional powers. PBRCH is initially TRUE.

9. A listing of the file SOLVE.EQN should reveal how additional inverse functions can be added to the package.

10. File MATRIX.ARR contains a matrix division operation that can be used to solve simultaneous linear algebraic equations.

9.6 ARRAY.ARI: ARRAY OPERATIONS

Purpose: File ARRAY.ARI provides a facility for establishing generalized arrays, for extracting or assigning to their components, and for performing elementwise operations between arrays or between arrays and scalars.

Prerequisite File: ARITH.MUS
Also helpful is file LOG.ALG if elements involve logarithms, or files TRGPOS.ALG and TRGNEG.ALG if elements involve trigonometric functions.

Usage:

{expr1, expr2, ..., exprN},	a column vector
[expr1, expr2, ..., exprM],	a row vector
array1 operator array2,	array-array operations
scalar operator array,	scalar-array operations
function (array),	function applications
array rowvector,	extraction of components
name rowvector: expr.	assignment of an element

Examples:

```
? [3, X] + [5, X, Y];                  % Array addition operation %
@: [8, 2*X, Y]

? 2 * {X, LN(Y)};                      % Scalar multiplication %
@: {2*X,
    2*LN(Y)}

? {[4, X, 6], [X^2, 4, 2*X]} / 2;      % Scalar division %
@: {[2, X/2, 3]
    [X^2/2, 2, X]}

? SIN ([X, Y]);                        % Application of function %
@: [SIN(X), SIN(Y)]                    % Requires TRGPOS.ALG loaded %

? A: [X, [Y,Z], [W]]$

? A [2];                               % Extraction of an element %
@: [Y,Z]

? A [2,1];                             % Extraction of an element %
@: Y

? A [2] [1];                           % Extraction of an element %
@: Y

? A [3]: SIN(X)$                       % Assignment to an element %

? A;
@: [X, [Y,Z], SIN(X)]
```

Remarks:

1. An array is either a column or a row vector. Each element of the vector can be any arbitrary expression, which includes being another vector. Thus a two dimensional array can be represented as a vector of vectors. Naturally arrays can be nested to any desired depth.

2. Column vectors are printed starting each element on a new line. Thus, 2-dimensional arrays generally look better as a column of rows than as a row of columns. Higher dimensional arrays generally appear best as a column of rows of rows . . . of rows.

3. When rows or columns of unequal length are combined elementwise by an arithmetic operation, the shorter of the two arrays is treated as having implied zeros corresponding to the extra elements of the longer array. (Consistent with this interpretation, a subscript value larger than the number of explicit elements in a row or column yields a zero as the value of the element.) Thus, upper-triangular, left-triangular, and other "ragged" arrays are efficiently represented.

4. When an array is combined with a scalar, the latter distributes over the elements of the array.

5. Functions of one argument that employ the general rule-application function named SIMPU distribute over the elements of an array. For example SIN ([X,Y]) —> [SIN(X), SIN(Y)].

6. Subscripts can be recursively employed to any level, and they can be symbolic. For example, [Y, Z][2][N] —> Z[N].

7. Comments in file ARRAY.ARI indicate how to save space by omitting the column, subscript, or element-assignment packages. (Rows together with FIRST, REST, SECOND, etc. are sufficient for many purposes.)

8. File MATRIX.ARR implements matrix operations on arrays, including matrix transpose, multiplication, division, and power, including inverse.

9. Some terminal character sets do not include brackets (ie "[" and "]") and/or braces (ie "{" and "}") normally used to delimit row and column vectors. If your terminal does not have brackets, the muSIMP parser will accept the pair of characters "<<" and ">>" in place of "[" and "]" respectively. As soon as such a pair is typed into muSIMP, that pair of characters will also be used on output to replace the corresponding bracket. Similarly, "(<" can be used in place of "{", and ">)" in place of "}".

   For example, the **APPLE][** screen supports brackets but not braces. Thus users will propably want to stick with brackets for row vectors, but will have to use "(<" and ">)" for column vectors. See Section 13.17 on how to enter the "[" character.

9.7 MATRIX.ARR: MATRIX OPERATIONS

Purpose: File MATRIX.ARR provides the following matrix operations on arrays: transpose, multiplication, division, inverse, and other integer powers, together with determinants. Elementwise operations such as addition are provided by the prerequisite file ARRAY.ARI.

Prerequisite File: ARRAY.ARI

Usage:

IDMAT (positiveinteger),	identity matrix
array `,	matrix transpose
array1 . array2,	matrix dot product
array1 \ array2,	matrix division
array ^ integer,	matrix powers
DET (array).	determinant

Examples:

```
? IDMAT(2);
@: {[1],
    [0, 1]},

? A: {[1, 2], [0, 3]}$  B: {P, 6}$

? B`;
@: [P, 6]

? A`;
@: [{1,
     2}, {0,
          3}]

? A` . IDMAT(2);
@: {[1, 0],
    [2, 3]}

? A . B;
@: {P+12,
    18}

? A \ B;
@: {P-4,
    2}

? A ^ 2;
@: {[1, 8],
    [0, 9]}

? A ^ -1;
@: {[1, -2/3],
    [0, 1/3]}

? DET(A);
@: 3
```

Remarks:

1. The function **IDMAT(num)** returns a left-triangular identity matrix of dimension equal to positive integer argument.

2. The postfix **matrix transpose** operator "`" has a left binding power of 170. This "backward accent" character, ASCII code 60 hex, different from an apostrophe or single quote, is often found on the same key as the "@" character. If your terminal does not provide this character, "**TR**" can be used as the transpose operator instead. For instance, **APPLE][** users using the standard APPLE keyboard will have to use "TR".

3. The transpose of a scalar is a scalar, the transpose of a row is the column of the transposes of its elements, and the transpose of a column is the row of the transposes of its elements. These rules are recursively employed so that the transpose of a ragged and/or nested matrix is appropriately performed. These rules also convert a column of rows into a row of columns, which does not print attractively. However, multiplication by an appropriate sized identity matrix always yields the attractive column-of-rows form of a matrix.

4. The matrix **dot product** infix operator "." has left and right binding powers of 125. The dot product is recursively defined as follows:

```
scalar.expr  —>  scalar*expr

    row.col  —>  row1.col1 + row2.col2 + ... + rowN.colN

    col.row  —>  {[col1.row1, col1.row2, ...],
                  [col2.row1, col2.row2, ...],
                        .          .       .
                  [colN.row1, colN.row2, ...]}

  rowA.rowB  —>  [[rowA1.rowB1, rowA1.rowB2, ...],
                  [rowA2.rowB1, rowA2.rowB2, ...],
                        .          .       .
                  [rowAN.rowB1, rowAN.rowB2, ...]]

  colA.colB  —>  {{colA1.colB1,
                   colA1.colB2,
                        .
                   colA1.colBN},
                  {colA2.colB1,
                   colA2.colB2,
                        .
                   colA2.colBN},
                        .
                  {colAN.colB1,
                   colAN.colB2,
                        .
                   colAN.colBN}}
```

5. Consistent with the interpretation described in Section 9.6, when a row and column are of unequal length, the shorter is treated as having implied trailing zero elements when forming the dot product of the two. These interpretations of matrix products are recursively employed so that matrix products of nested and/or ragged arrays are appropriately performed.

6. Raising a matrix to a positive integer power is performed by repeated use of the dot product. The matrix inverse is indicated by raising it to the -1 power. Thus for a matrix A:

```
A^0   —> IDMAT (LENGTH(A)-1)
A^1   —> A
A^-1  —> A inverse
```

For integer n > 1:

```
 A^n  —> A . (A ^ (n-1))
A^-n  —> (A ^ -1) . (A ^ (n+1))
```

When a matrix is singular, raising it to a negative power yields warning messages about divisions by zero, and the offending subexpressions are encapsulated in a question mark form according to the usual muMATH computational error treatment.

7. The **matrix division** operator is the back-slash character: "\. It is an infix operator with a left and right binding power of 125. As explained in Section 4.1, it may be necessary to change all occurrences of "\" to "MDV" in the file **MATRIX.ARR** if the back-slash is not included in your terminals character set. **APPLE][** SoftCard users can type **CTRL-B** to get a back-slash.

8. If **A** is a square, nonsingular matrix, then

```
A\B
```

is equivalent to

```
(A^-1) . B
```

Since matrix division is the more efficient of the two methods, we strongly recommend using the division operator unless the inverse of **A** is required for subsequent calculations. In addition, provided **B** is consistent, the division operator will yield a parameterized solution even when **A** is singular. In this event, the parameters will be designated by the arbitrary value forms: ARB(1), ARB(2), ...

9. Consistent with the definition of ragged arrays, if the array **A** has more rows than columns, DET(A) is 0. However, if **A** has more columns than rows, DET(A) is the determinant of its left square sub-matrix. This is a useful generalization of the conventional definition of determinants.

10. Comments in file **MATRIX.ARR** indicate how to save space by omitting the matrix transpose, division, power and/or determinant packages.

9.8 LOG.ALG: LOGARITHMIC SIMPLIFICATION

Purpose: File LOG.ALG provides facilites for logarithmic simplifications.

Prerequisite File: ALGEBRA.ARI

Control Variables:

1. **LOGBAS**, which is the default LOGarithm BASe when LOG is given only one argument. LOGBAS is initially #E, the base of the natural logarithms.

2. **PBRCH** controls whether multiply branched logarithmic functions are automatically simplified by Picking a particular BRanCH. It is initially and normally TRUE.

3. **LOGEXPD**, which controls expansion or collection of logarithms, and base conversion. It is initially 0.

Usage:

LN (expr),
LOG (expr),
LOG (expr, base),
LOGEXPD (expr, ControlValue).

Examples:

```
? LN (#E^X^2);
@: X^2

? LOGEXPD: 15$

? LN (#E*R*S^3/T);
@: 1 + LN(R) + 3*LN(S) - LN(T)

? LOGEXPD: 2$

? LOG (X, Y) / LOG (X);
@: 1 / LN(Y)
```

Remarks:

1. Since the emphasis of muMATH is on exact results, there is no attempt to approximate irrational logarithms. The Taylor Series can be used to find an approximation if desired. See Section 9.14 describing TAYLOR.DIF for details.

2. The unbound variable **#E** denotes and possesses the properties associated with the base of the natural logarithms.

3. Although all logarithms are stored internally as functions of two arguments, LN(expr) is used as an abbreviation for

LOG(expr,#E) on input and output.

4. LOG(expr) is used as an abbreviation for LOG(expr,**LOGBAS**) on input and output, where LOGBAS is a control variable initially set to #E. Some users may prefer to set LOGBAS to ten (10) if the problem involves common logarithms.

5. The following simplification always occurs:
   base ^ LOG(expr,base) --> expr.

6. If the control variable PBRCH is TRUE, the following simplifications will occur:
   LOG (1, base) --> 0,
   LOG (base, base) --> 1,
   LOG (base^expr, base) --> expr.

7. If LOGEXPD is a positive integer multiple of 2 and base is not equal to LOGBAS, the following transformation will occur:

   LOG (expr, base) --> LOG (expr, LOGBAS) / LOG (base, LOGBAS).

   If LOGEXPD is a negative multiple of 2, the opposite transformation of combining appropriate products or ratios of logarithms occurs.

8. If LOGEXPD is a positive integer multiple of 3, the following transformation will occur:

   LOG (expr^exp, base) --> exp * LOG (expr, base).

   If LOGEXPD is a negative multiple of 3, the opposite transformation will occur.

9. If LOGEXPD is a positive multiple of 5, the following transformation will occur:

LOG (expr1*expr2, base) --> LOG (expr1, base) + LOG (expr2, base),
LOG (expr1/expr2, base) --> LOG (expr1, base) - LOG (expr2, base).

   If LOGEXPD is a negative multiple of 5, the opposite transformation (i.e. collecting sums of logs) will occur.

10. If LOGEXPD is a positive multiple of 7 and base is not equal to LOGBAS, the following transformation will occur:

    base ^ nonnumber --> LOGBAS ^ (nonnumber * LOG(base,LOGBAS)).

11. The function LOGEXPD(expr,integer) evaluates its first argument after temporarily setting the control variable LOGEXPD to the value of the second argument. This provides a convenient means of seeing the effect various values of LOGEXPD has on an expression.

9.9 TRGPOS.ALG: TRIGONOMETRIC SIMPLIFICATION, POSITIVE TRGEXPD

Purpose: TRGPOS.ALG provides the following trig transformations:

1. exploitation of symmetry to simplify trig arguments,
2. replacement of other trig functions by sines and cosines,
3. replacement of integer powers of sines and cosines by linear combinations of sines and cosines of multiple angles,
4. replacement of products of sines and cosines by linear combinations of sines and cosines of angle sums,
5. replacement of integer powers of sines by those of cosines or vice-versa.

Prerequisite File: ALGEBRA.ARI

Loading TRGNEG.ALG after TRGPOS.ALG preserves the full capabilities of both files. Loading TRGPOS.ALG after TRGNEG.ALG destroys the angle-reduction capabilities of the latter, thus saving some space.

Control Variables:

1. **TRGEXPD** controls replacement of trig functions by sines and cosines and replacement of powers and products of sines and cosines by linear combinations. Only positive values of TRGEXPD are significant when TRGPOS.ALG is loaded without TRGNEG.ALG.

2. **TRGSQ** controls the conversion of integer powers of sines to cosines and vice-versa.

Usage:

SIN (expr),
COS (expr),
TAN (expr),
CSC (expr),
SEC (expr),
COT (expr).

Examples:

```
? TRGEXPD: 2$

? TAN(A)*COS(A) + 1/CSC(A);
@: 2 * SIN(A)

? TRGSQ: 1$

? 2*COS(X)^2 + 1/CSC(X)^2;
@: 1 + COS(X)^2

? TRGEXPD: 15$

? COS(X)^2*SIN(X);
@: SIN(X)/4 + SIN(3*X)/4
```

Remarks:

1. The following simplifications always occur:

SIN(0)	-->	0	COS(0)	-->	1
SIN(-X)	-->	-SIN(X)	COS(-X)	-->	COS(X)

2. When TRGEXPD is a positive multiple of 2, then tangents, cotangents, secants, and cosecants are replaced by corresponding expressions involving sines and/or cosines. For example, when TRGEXPD = 2, CSC(X) --> 1/SIN(X).

3. When TRGEXPD is a positive multiple of 3, then integer powers of sines and cosines are expanded in terms of sines and cosines of multiple angles. For example, when TRGEXPD = 3,

   COS(X)^2 --> (1+COS(2*X))/2.

   These transformations usually give the most attractive results if NUMNUM and perhaps also DENNUM are positive multiples of 6.

4. When TRGEXPD is a positive multiple of 5, then products of sines and cosines are expanded in terms of angle sums. For example, when TRGEXPD is 5,

   SIN(X)*SIN(Y) --> (COS(X-Y) - COS(X+Y))/2.

   These transformations usually give the most attractive results if NUMNUM is a positive multiple of 30 and DENNUM is a positive multiple of 2.

5. Expanding an expression over a common denominator with TRGEXPD = 30 yields a normal form for a large class of rational trigonometric expressions. Thus, the most straightforward way to prove most trig identities is to evaluate the difference of the two sides of the identity with TRGEXPD: NUMNUM: DENDEN: 30, PWREXPD: 6, and DENNUM: -30.

6. TRGEXPD = 30 has the effect of "linearizing" trigonometric polynomials, thus facilitating harmonic or Fourier analysis.

7. For integer n with |n| > 1 and for all u, when TRGSQ is a positive integer, then

   COS(u)^n --> COS(u)^MOD(n,2) * (1 - SIN(u)^QUOTIENT(n,2))^2.

   Conversely, when TRGSQ is a negative integer, then

   SIN(u)^n --> SIN(u)^MOD(n,2) * (1 - COS(u)^QUOTIENT(n,2))^2.

   These transformations are sometimes useful for transforming a trigonometric polynomial to a more compact form.

8. Even when a trig polynomial is preferred for the final form, net simplification is often achieved by evaluating with TRGEXPD = 30, then TRGEXPD = -30, then perhaps again with TRGSQ = 1 or TRGSQ = -1 according to the appearance of the result produced by TRGEXPD = -30.

9. File TRGNEG.ALG provides for the negative settings of TRGEXPD to yield the converse of the above transformations.

9.10 TRGNEG.ALG: TRIGONOMETRIC SIMPLIFICATION, NEGATIVE TRGEXPD

Purpose: TRGNEG.ALG provides the following trig transformations:

1. exploitation of symmetries to simplify trig arguments,
2. angle reduction,
3. multiple-angle expansion,
4. angle-sum expansion,
5. elimination of reciprocals of trigonometric forms,
6. elimination of certain products of trigonometric forms,
7. simplification of trig functions of their own inverses,
8. replacement of sines and cosines by complex exponentials.

Prerequisite File: ALGEBRA.ARI

Loading TRGPOS.ALG after TRGNEG.ALG destroys the angle-reduction capabilities of the latter, thus saving some space. Loading TRGNEG.ALG after TRGPOS.ALG preserves the full capabilities of both files.

Control Variables:

1. **TRGEXPD** controls the use of multiple angle and angle sum expansions and replacement of trig functions by complex exponentials. Only negative values of TRGEXPD are significant when TRGNEG.ALG is loaded without TRGPOS.ALG.

Usage:

SIN (expression),
COS (expression),
TAN (expression),
CSC (expression),
SEC (expression),
COT (expression),
TRGEXPD (expression, integer).

Examples:

```
? SIN (20*#PI/7);
@: SIN (#PI/7)

? SIN (7*#PI/3);
@: 3^(1/2)/2

? SIN (ASIN(X+5));
@: X + 5

? TRGEXPD (SIN (2*X + Y), -15);
@: 2*COS(X)^2*SIN(Y) + 2*COS(X)*COS(Y)*SIN(X) - SIN(Y)

? FCTR (TRGEXPD (SIN (X), 7));
@: #I * (1/#E^(#I*X) - #E^(#I*X)) / 2
```

Remarks:

1. Since the emphasis of muMATH-80 is on exact results, there is no attempt to approximate irrational trig expressions. However, they can be approximated using a series expansion.

2. The ratio of the circumference to the diameter of a circle is represented by the unbound variable #PI. The user is of course free to assign a rational approximation to #PI.

3. Angles are assumed to be measured in radians. Those who prefer some other unit such as degrees may wish to define additional functions named SIND, COSD, etc.

4. Sines and cosines of angles which are numeric multiples of #PI are reduced to equivalent sines or cosines in the range of 0 to #PI/4 radians. Then sines and cosines of the special angles 0, #PI/6, and #PI/4 are evaluated exactly. (See examples above.)

5. Symmetry is exploited to simplify the arguments of sines and cosines. For example,

```
SIN (-X)  -->  -SIN (X),
COS (-X)  -->  COS (X).
```

6. Trigonometric functions of the corresponding inverse trig functions simplify. The inverse trig functions are named ATAN, ASIN, ACOS, ACOT, ACSC, and ASEC.

7. Products of a tangent, cotangent, secant, or cosecant with another trig function of the same argument are simplified to 1 or to a single form where possible. For example,

```
SEC(X)*COS(X)  -->  1,
TAN(X)*COS(X)  -->  SIN(X).
```

   For an expression such as SEC(X)^2*COS(X)^2 it is necessary to reevaluate with EXPBAS being a negative multiple of 2 in order to achieve the desired trig transformation.

8. When TRGEXPD is a negative multiple of 2, then negative powers of tangents, cotangents, secants, and cosecants are replaced by corresponding positive powers of the corresponding reciprocal trig functions. For example, when TRGEXPD = -6,

```
1/TAN(X+7)^3 --> COT(X+7)^3.
```

   For technical reasons, negative powers of sines and cosines are treated in file TRGPOS.ALG.

9. When TRGEXPD is a negative multiple of 3, then sines and cosines of mulitple angles are expanded in terms of sines and cosines of non-multiple angles. For example, when TRGEXPD = -6,

```
SIN (2*X)  -->  2*SIN(X)*COS(X),
COS (3*X)  -->  4*COS(X)^3 - 3*COS(X).
```

   These transformations usually give the most attractive results if NUMNUM is a positive multiple of 6.

10. When TRGEXPD is a negative multiple of 5, then sines and cosines of angle sums and differences are expanded in terms of sines and cosines of nonsums and nondifferences. These transformations usually give the most attractive results if NUMNUM is a positive multiple of 6.

11. When TRGEXPD is a POSITIVE multiple of 7, then sines and cosines are converted to complex exponentials. For example, when TRGEXPD = 14, then COS(X) --> (#E^(#I*X) + 1/#E^(#I*X)) / 2. The opposite transformation, provided in file ARITH.MUS, is requested when TRGEXPD is a negative multiple of 7. A worthwhile net trig simplification can sometimes be achieved by converting to complex exponentials, expanding or factoring judiciously, then converting back to trig functions.

12. In muMATH-80 changing the value of an option variable does not affect the values of expressions that have already been evaluated. Thus, after changing the value of TRGEXPD and other relevant variables it may be necessary to use EVAL to get the desired effect.

13. Function TRGEXPD reevaluates its first argument with TRGEXPD temporarily set to the value of the second argument. Thus, it provides a convenient way to accomplish a trigonometric transformation without the necessity of altering the global setting of the TRGEXPD control variable.

14. File TRGPOS.ALG has other important trig transformations, many of which are the opposite of those provided in file TRGNEG.ALG. Generally, the positive settings yield a more canonical (but not necessarily more compact) representation. A net simplification is often achieved by evaluating an expression with the relevant option variables set positive, then reevaluating with them set the other way. Thus, files TRGPOS.ALG and TRGNEG.ALG comprise an important complementary pair of files. Since together the files are relatively large, for some applications it may be desirable to extract and combine a few of the required features from both files, together perhaps with a few additional transformations modeled after them.

9.11 DIF.ALG: SYMBOLIC DIFFERENTIATION

Purpose: File DIF.ALG provides facilities for finding the first partial derivative of an expression with respect to a variable.

Prerequisite File: ALGEBRA.ARI
If the expression to be differentiated contains transcendental functions, the appropriate file should also be loaded.

Usage:
DIF (expression, variable).

Examples:

```
? DIF (A*X^2 - 3*X + 2, X);
@: 2*A*X - 3

? DIF (LN(3*X^2+A*X), X);
@: (6*X+A) / (3*X^2+A*X)

? DIF (#E^X*TAN(X)/X, X);
@: -#E^X*TAN(X)/X^2 + #E^X*SEC(X)^2/X + #E^X*TAN(X)/X

? DIF (F(X), X);
@: DIF (F(X), X)

? DIF (Y, X);
@: 0;

? DIF (DIF(SIN(X*Y),X), Y);
@: -X*Y*SIN(X*Y) + COS(X*Y)
```

Remarks:

1. When the differentiation rule for a function or operator is not known to the system:

   a) The derivative is 0 if none of the arguments or operands contain the differentiation variable. For example,
   DIF (F(Y), X) --> 0.

   b) The derivative is not evaluated otherwise. For example,
   DIF (F(X), X) —> DIF (F(X), X).

2. If you extend muMATH with the addition of new mathematical functions or operators, study of file DIF.ALG will reveal how to add new differentiation rules for the new functions or operators.

3. The differentiation "variable" can actually be an arbitrary expression, which is then treated the same as a simple variable for differentiation purposes. (This is occasionally quite

useful, such as when performing a square-free factorization or when deriving the Euler-Lagrange equations for a specific variational calculus problem.)

4. Higher-order partial derivatives can be requested directly by nested use of DIF (see above example). However, repeated differentiation can require dramatically increasing time and space, especially for products, quotients, and composite expressions.

5. The utility function **FREE** (expr, indet) is a predicate which returns TRUE if and only if expr is free of (i.e. contains no occurrences of) indet.

9.12 INT.DIF: SYMBOLIC INTEGRATION

Purpose: INT.DIF provides indefinite symbolic integration.

Prerequisite File: DIF.ALG

Usage:

INT (expression, variable).

Example:

```
? INT (6*A*X^2 + 3*X*B - 4, X);
@: 3*A*X^3 + 3*B*X^2/2 - 4*X

? INT (X*SIN(X^2), X);
@: - COS(X^2)/2

? INT (X/(A*X^2+B), X);
@: LN(A*X^2+B) / (2*A)
```

Remarks:

1. When INT is unable to determine a closed-form integral of portions of an expression, the returned expression will contain unevaluated integrals of those portions. For example,
   INT (X + A*#E^X/X, X) —> X^2/2 + A*INT(#E^X/X,X).

2. INT uses distribution over sums, extraction of factors that do not depend upon the integration variable, known integrals of the built-in functions, a few reduction rules, and a "derivatives-divides" substitution rule. Consequently, integration succeeds only for a relatively modest class of integrands. However:
   a) The class is large enough to be quite useful,
   b) File **INTMORE.INT** contains additional rules,
   c) Integration of a truncated series approximation of an integrand, perhaps derived using file TAYLOR.DIF, can often yield a truncated series representation of otherwise intractable integrals.

3. A careful study of files INT.DIF and INTMORE.INT will reveal how to add additional integration rules to the basic package.

4. The integration "variable" can be an arbitrary expression, which is then treated the same as a simple variable for integration purposes.

5. Successful integration may depend upon the form of the integrand, after it is simplified according to the current flag settings. Generally speaking, it is best to employ conservative flag settings which do relatively little to alter the form of an expression. INT will automatically expand, factor, employ trigonometric transformations, etc. as necessary in an attempt to find the integral of an expression.

9.13 INTMORE.INT: EXTENDED SYMBOLIC INTEGRATION

Purpose: File INTMORE.INT extends the facilities provided by INT.DIf to find the indefinite integral of a mathematical expression. In addition the capability to find **definite** integrals is provided.

Prerequisite File: INT.DIF
LIM.DIF is also required for finding "improper" definite integrals containing infinite limits or finite limits at which the indefinite integral has an "indeterminate" form such as 0/0.

Usage:
INT (expr, variable),
DEFINT (expr, variable, lowerlimit, upperlimit).

Example:

```
? INT (X*SIN(X)^2, X);
@: -X*SIN(2*X)/4 + X^2/4 - COS(2*X)/8

? INT (X*LN(A*X), X);
@: X^2*LN(A*X)/2 - X^2/4

? DEFINT (A*X^2, X, 0, 1);
@: A/3

? DEFINT (1/X^2, X, 1, PINF);                    % Requires LIM.DIF %
@: 1 + MZERO
```

Remarks:

1. If file LIM.DIF has been loaded, then DEFINT uses the difference in the limits of the indefinite integral as the integration variable approaches the two integration limits. Otherwise, DEFINT merely uses substitution into the indefinite integral, which is appropriate only for proper integrals.

2. When DEFINT is unable to determine a closed-form integral, the unevaluated integral is returned. For example,
   DEFINT (X+A*#E^X/X, X, 0, 1) —>
   DEFINT (X+A*#E^X/X, X, 0, 1).

3. Nested integration can be used to request directly an iterated integration, such as occurs for appropriate multiple-integrations. For example, to integrate the expression y*x^2 over the upper unit semi-disk, we could evaluate
   DEFINT (DEFINT(Y*X^2,Y,0,(1-X^2)^(1/2)), X, -1, 1).

   However, the class of expressions which is repeatedly integrable is dramatically smaller than the class which is once integrable.

4. Section 9.12 contains other appropriate remarks concerning integration.

9.14 TAYLOR.DIF: TAYLOR SERIES EXPANSION

Purpose: File TAYLOR.DIF provides a function that uses repeated differentiation and substitution to yield a truncated Taylor series expansion of an expression with respect to one of the unbound variables therein. Among the many applications of Taylor series are:

1. to reveal qualitative information about functions defined by complicated expressions;
2. to replace complicated expressions by simpler symbolic approximations in order to permit comprehensible further analytic operations;
3. to derive efficient formulas for the approximate numerical evaluation of complicated expressions.

Prerequisite Files: DIF.ALG
Also desirable is file LOG.ALG if the expression contains logarithms, file TRGPOS.ALG and/or TRGNEG.ALG if the expression contains trigonometric functions, etc.

Usage:
TAYLOR (expr, variable, expansionpoint, degree).

Example:

```
? TAYLOR (#E^X, X, 0, 5);
@: 1 + X + X^2/2 + X^3/6 + X^4/24 + X^5/120

? TAYLOR (#E^SIN(X), X, 0, 6);
@: 1 + X + X^2/2 - X^4/8 - X^5/15 - X^6/240
```

Remarks:

1. The requested degree must be a nonnegative integer.

2. Expressions might not have truncated Taylor series expansions of the desired degree. This is generally revealed by a singularity such as an attempted division by 0, which produces the warning followed by encapsulated subexpressions such as ?(1/0).

3. The expression must be differentiable "degree" number of times. Naturally the differentiation rules of all subexpressions must be established.

4. Since the complexity of successive derivatives can grow dramatically with each iteration, the time and space required to compute the Taylor Series expansion of an expression will also grow correspondingly.

5. Although TAYLOR attempts to produce its result in a series-like form, the user may have to reevaluate the result with appropriate settings of control variables such as PWREXPD, NUMNUM, DENNUM, etc., in order to obtain the most attractive result.

6. The Taylor Series can often be used to generate other types of truncated series by making an appropriate substitution, using Taylor, then making the inverse substitution. For example, the substitution X --> 1/Y yields an expansion in nonpositive powers, whereas the substitution X --> #E^(#I*Y) yields a Fourier series.

7. There are more complicated but more efficient methods for computing Taylor-series and other series expansions. For example, see Knuth [The Art of Computer Programming, Vol 2], Zippel [in the Proceedings of the 1976 ACM Conference on Symbolic and Algebraic Computation].

9.15 LIM.DIF: LIMITS OF FUNCTIONS

Purpose: File LIM.DIF provides facilities for finding the one-sided limit of a mathematical expression as one of its variables approaches a value.

Prerequisite File: DIF.ALG
LOG.ALG is also advisable, as are TRGPOS.ALG and TRGNEG.ALG to diminishing extents.

Usage:
LIM (expression),
LIM (expression, variable),
LIM (expression, variable, point),
LIM (expression, variable, point, TRUE).

Examples:

```
? LIM (SIN(X)/X, X);
@: 1

? LIM ((X^2-4*X+3) / (2*X^2-13*X+21), X, 3);
@: -2

? LIM (((2-X)*#E^X-X-2) / X^3, X, 0);
@: -1/6

? LIM (PINF - MINF + 5);
@: PINF

? LIM (1/X, X, 0);                    % APPROACH FROM THE RIGHT %
@: PINF

? LIM (1/X, X, 0, TRUE);              % APPROACH FROM THE LEFT %
@: MINF
```

Remarks:

1. **LIM** is a function which returns the limit of its first argument as its second argument approaches the third argument. An optional fourth argument is used to determine the direction of approach. If it is FALSE or not given, the limit is from the right (i.e. the second argument decreases toward the third). If non-FALSE, the limit is from the left.

2. Besides ordinary numeric or nonnumeric values, the third argument of LIM can be PINF or MINF, designating plus infinity and minus infinity respectively.

3. A question-mark returned by the LIM function indicates the nonexistence of a unique one-sided limit. For example,
   LIM (SIN(X), X, MINF) --> ?.

4. LIM returns PZERO to represent 1/PINF, MZERO to represent 1/MINF, and CINF to represent 1/0, which designates "Complex INFinity": infinite magnitude in an arbitrary direction.

5. The default value for the third argument is 0. The second argument can also be omitted, in which case LIM is merely EVAL extended to simplify expressions containing PINF, MINF, CINF, PZERO, MZERO, and "?" — the special values that can be returned by previous uses of LIM.

6. LIM may query the user about the sign of various subexpressions or whether they represent ODD or EVEN integers, especially if the problem entails variables in addition to the limit variable. In order to save space, no great effort is made to avoid questions by making obvious deductions from previous answers. Thus, be patient with the naivete` of these queries: at least the same **exact** question will not be asked twice during the computation of one limit. In order to obtain the results corresponding to alternative answers, repeat the problem as necessary, choosing different combinations of answers to the queries each time.

7. LIM uses heuristics to transform the expressions and guide the choice between alternative forms of L'Hopital's rule for "indeterminate" forms. Since these techniques are not guaranteed to succeed, the depth of recursion on L'Hopital's rule is limited to the value of the global variable named #LIM, which is initially 3. Termination due to this depth limitation or due to the occurrence of a function that the package is not equipped to handle results in an answer containing one or more subexpressions of the form:
   LIM (expression, variable, 0).

   If the depth limitation is the cause, then applying EVAL to the answer will employ up to #LIM additional levels of L'Hopital's rule. Alternatively, the user may wish to increase #LIM, but this increases the hazard of storage exhaustion, so the user may wish to decrease #LIM instead.

8. To verify that a limit is actually two-sided, compare the values obtained from above and below.

9. The end of file LIM.DIF provides examples of how additional limit rules can be added in a manner similar to those for LOG, ATAN, SIN and COS.

9.16 SIGMA.ALG: CLOSED-FORM SUMMATION AND PRODUCTS

Purpose: File SIGMA.ALG provides facilities for determining closed-form sums and products. Using classic capital sigma and capital pi notation, these two functions are defined as

$$\text{SIGMA}\ (u_j,\ j,\ m,\ n) = \sum_{j=m}^{n} u_j = u_m + u_{m+1} + \ldots + u_n,$$

$$\text{PROD}\ (u_j,\ j,\ m,\ n) = \prod_{j=m}^{n} u_j = u_m * u_{m+1} * \ldots * u_n\ ,$$

where u_j is an expression which can depend upon j. For SIGMA, m and/or n can be symbolic expressions rather than mere integers.

Prerequisite Files: ALGEBRA.ARI
File LOG.ALG should also be included if the expression contains logarithms, file TRGPOS.ALG and/or TRGNEG.ALG if the expression contains trig functions, etc. File LIM.DIF is necessary for evaluating sums having an infinite limit.

Usage:

SIGMA (expr, J, M, N)	the summation from J = M to N,
PROD (expr, J, M, N)	the product from J = M to N.

Examples:

```
? SIGMA (A*J, J, 1, N);
@: A*N*(N+1)/2

? SIGMA (1/K - 1/(K+1), K, 1, N);
@: N / (1+N)

? SIGMA (2^-J, J, 0, PINF);              % Requires LIM.DIF %
@: 2 + MZERO

? PROD (LN(J), J, 2, 4);
@: LN(2) * LN(3) * LN(4)
```

Remarks:

1. Iteration is used when both limits are numeric. Otherwise the problem is analogous to symbolic integration.

2. Implemented techniques include checking for telescoping sums, distributing summation over the terms of a summand, extracting factors that do not depend upon the index, and using known rules for sums of exponentials, powers, etc.

3. When a lower limit exceeds the upper limit, a sum is 0 or a product is 1.

4. Study of SIGMA.ALG will indicate how additional summation or product rules can be added to the package.

5. In order to evaluate sums having infinite limits the LIM.DIF package should also be loaded. Positive infinity is then denoted by PINF and negative infinity by MINF.

6. There are more comprehensive and complicated summation and product algorithms which handle a larger class of expressions. They are discussed in the papers by Gosper and Moenck in the Proceedings of the 1977 MACSYMA User's Conference [NASA CP2021].

10. HOW TO LEARN MORE ABOUT COMPUTER ALGEBRA

We expect that many who experience muMATH or another computer algebra system will want to learn more about this fascinating subject and will want to introduce it to others who may lack an appropriate computer to run muMATH. Accordingly, here is a brief guide to the relevant professional society, the literature, and some widely available systems.

10.1 THE PROFESSIONAL SOCIETIES

The Association for Computing Machinery **S**pecial **I**nterest **G**roup on **S**ymbolic and **A**lgebraic **M**anipulation (SIGSAM) is the major international professional society for computer algebra. The **SIGSAM Bulletin** is the most concentrated and up-to-date source for abstracts and working papers together with announcements of meetings and systems. For information about joining, including special student rates, write the ACM at 1133 Avenue of the Americas, New York, NY 10036.

Some European groups devoted to computer algebra are:

1. SAM-AFCET: Contact M. Bergman, Faculte des Sciences de Luminy, Case 901, 13009 Marseille, France; or contact J. Calmet, Universitat Karlsruhe, Institut fur Informatik I, 75 Karlsruhe 1, Postfach 6380, West Germany.

2. NIGSAM: Contact Y. Sundblad, Department of Numerical Analysis and Computer Science, KTH S-10044 Stockholm, Sweden.

3. SEAS/SMC: Contact J.A. van Hulzen, Twente University of Technology, P.O. Box 217, 7500 AE Enschede, The Netherlands.

10.2 THE LITERATURE

Regrettably, there is not yet a textbook devoted to computer algebra, and the few textbooks that contain relevant material are rather advanced. Most of the information is sparsely scattered in research journals or less accessible conference proceedings and reports. However, the following relatively accessible references contain surveys, bibliographies, and collections of articles that should serve as a good point of departure for exploring most facets of the literature:

1. ACM SIGSAM Bulletin, ACM, New York, all issues.

2. Communications of the ACM 14, No. 10, August 1971, entire issue.

3. SIAM Journal on Computing 8, No. 3, August 1979, entire issue.

4. Communications of the ACM 9, No. 10, August 1966, entire issue.

5. Journal of the ACM 18, No. 4, October 1971, entire issue.

6. E.W. Ng, editor, **Symbolic and Algebraic Computation,** Lecture Notes in Computer Science, 72, Springer-Verlag, New York, 1979.

7. R.D. Jenks, editor, **Proceedings of the 1976 ACM Symposium on Symbolic and Algebraic Computation,** ACM, New York, 1976.

8. V.E. Lewis, editor, **Proceedings of the 1979 MACSYMA User's Conference,** M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, Massachusetts, 1979.

9. C.M. Anderson, editor, **Proceedings of the 1977 MACSYMA User's Conference,** NASA CP-2012, 1977.

10. Knuth, D.E., **The Art of Computer Programming, Volume II, Seminumerical Algorithms,** Addison-Wesley, Reading, Mass., 1980.

11. Yun, D.Y.Y., and Stoutemyer, D.R., "Symbolic Mathematical Computation", **Encyclopedia of Computer Science and Technology, Volume 15 Supplement,** J. Belzer, A.G. Holzman and A. Kent, editors, M. Dekker, New York & Basel, 1980, pp. 235-310.

10.3 WIDELY AVAILABLE SYSTEMS

Despite an almost total lack of publicity, computer algebra has been available for large mainframe computers since 1951 when the first symbolic differentiation program was written. Since then there have been numerous major general-purpose computer algebra systems implemented for large computers. muMATH was developed in order to bring a general purpose computer algebra system to inexpensive personal computers.

Naturally, large mainframe computer algebra systems can accommodate larger expressions than muMATH and treat them more quickly — at least in terms of CPU time as opposed to **turn-around** time. However, some of the large systems are noninteractive, thus preventing exploratory dialogues familar to muMATH users. **Interactive systems** permit the user to enter a sequence of mathematical expressions and/or function definitions and simplification rules from the terminal. Each intermediate result can be viewed before deciding how to proceed. Interaction is less crucial for very large well-defined problems, but it is highly desirable for "one of a kind" problems or moderate-sized problems that could be done manually with some effort. Interactive systems are also far more motivating for educational purposes, where the problems tend to be small, numerous and varied.

In approximate order of increasing memory requirements, here are six of the most widely available general-purpose programs or systems that are currently supported:

1. **PICOMATH-80**^{tm} is a package of four computer-algebra demonstration programs written in simple BASIC. PICOMATH was developed by The Soft Warehouse to provide a computer-algebra exposure to those having a computer that is too small or of the wrong type to utilize any of the systems listed below. Though not comprehensive enough to be called a **system,** the four programs collectively span a large

enough class of expressions to be useful and to provide an inexpensive introduction to computer algebra. Since each program can fit into about 4 kilobytes on typical microcomputer BASIC implementations, PICOMATH can be installed on virtually any computer. Machine-readable versions of PICOMATH are distributed for many popular personal computers by Programma International at 3400 Wilshire Blvd, Los Angeles, California 90010 — both mail-order and through most personal computer stores. Machine-readable versions for other computers may be available directly from hardware manufacturers. The manual, available separately from Programma, contains the BASIC program listings and an adaptation guide for various dialects of BASIC or other programming languages.

2. **muMATH-80**tm is an interactive system that runs on personal microcomputers based on the 8080, 8085, or Z80 microprocessors, provided they have at least 32 kilobytes of memory for user programs and data, together with the CP/Mtm, CDOStm, IMDOStm, or Radio Shack TRSDOStm disk operating system. Written by The Soft Warehouse, muMATH is distributed to end users, computer stores and hardware manufacturers by Microsoft at 10800 N.E. Eighth, Suite 819, Bellevue Washington 98004.

3. **SAC-2** is a non-interactive system which runs on any computer that can directly run a 1966 standard FORTRAN program of at least 120 kilobytes. Information about SAC-2 is available from Professor George Collins, Computer Sciences Department, University of Wisconsin, 1210 West Dayton St., Madison, Wisconsin 53706.

4. **FORMAC** runs on any IBM 360 or 370 that can accommodate a PL/I program of at least 150 kilobytes. FORMAC is semi-interactive on some operating systems. Information about FORMAC is available from Knut Bahr at GMD/IFV, D-6100, Darmstadt, West Germany.

5. **ALTRAN** is a non-interactive system which runs on any computer that can directly run a 1966 standard FORTRAN program of at least 270 kilobytes. Information about ALTRAN is available from the Computing Information Library, Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974.

6. **REDUCE** is an interactive system that runs on the IBM 360 or 370, DEC 10 or 20, Univac 1100 series, Control Data Cyber series, Burroughs 6700, and several other computers, requiring a minimum of about 350 kilobytes. For information about REDUCE, write Professor Anthony Hearn, Computer Science Department, University of Utah, Salt Lake City, Utah 84112.

Additional systems are announced in back issues of the ACM SIGSAM Bulletin.

11. COMPUTER ALGEBRA IN EDUCATION

It should be clear to anyone who has experienced a general-purpose computer-algebra system such as muMATH that they have enormous potential for use in education as well as research. Not only can computer algebra make computing more attractive to mathematically inclined students; computer algebra can make mathematics more attractive to computer enthusiasts. It provides a great opportunity for mutual reinforcement and cross motivation between math and computer education.

Personal computers are becoming so prevalent that students, engineers, scientists, and mathematicians will soon be using computer algebra extensively. It should not be more than a year or two before general-purpose computer algebra is available on pocket calculators, because:

1. National Semiconductor now makes a low wattage Z80 microprocessor.

2. There are already hand-held terminals with 64 kilobytes of low wattage memory.

3. There are already hand-held calculators with a sufficiently large low watttage liquid-crystals to display a reasonably large mathematical expression.

Eventually some enterprising manufacturer will surely marry these three technologies, producing a hand-held calculator capable of running muSIMP and muMATH. Thus, it behooves every math and computer science educator to explore how this revolutionary tool can be used to aid education.

It is undeniably true that most students are far more intrigued and motivated by the artificial intelligence and game playing applications of computers than by the accounting and numerical applications that currently account for most computer usage. Thus, it is advisable to exploit this strong preferential interest to help teach both mathematics and computer science. If more good math, science, and engineering students are attracted to computers and more good computer-oriented students are attracted to math, then more students will ultimately learn to use computers effectively for both numeric and nonnumeric purposes.

Computer algebra makes a highly motivating introductory computer programming course for math, science and engineering students. Computer algebra is also an ideal principal language for such students, because numbers and arithmetic comprise appreciably less than half of the kindergarden to calculus math curriculum. Moreover, the limited-precision integer and floating-point arithmetic typical of traditional programming languages is not the kind of arithmetic taught in this curriculum or used in everyday life.

Some educators will undoubtedly feel that computer algebra will cause algebraic skills to atrophy or prevent them from developing in the first place. Similar concerns were undoubtedly expressed about Arabic

numerals, multiplication tables, logarithms, Laplace transforms and pocket numerical calculators; but we have survived their convenience. The National Council for Teachers of Mathematics strongly supports the use of numerical pocket calculators in classrooms, and every reason for this support is even more true of computer algebra.

Automatic symbolic mathematics makes it possible for students to concentrate on basic mathematical concepts rather than spending an inordinate amount of time mechanically performing transformations. Computer algebra lets students explore such fundamental concepts as commutativity, associativity, groups and rings. Moreover, the extensive algebraic capabilities of computer algebra enables students to investigate larger examples than is otherwise practical. Patterns thus revealed may suggest useful theorems. Conjectured patterns thus violated provide counterexamples against false hypotheses. Thus, computer algebra can contribute to teaching **mathematical discovery.**

Existing computer-algebra systems can also make other educational contributions:

1. Trace packages can be used to let students witness each step of an algebraic simplification, rather than merely the final result.

2. The very fact that algebra and calculus can be automated should encourage average and poor math students that the flashes of inspiration characteristic of quick students are unnecessary for those operations -- there is revealed hope for the slower more methodical students.

3. For students who know how to program in the language in which the computer algebra packages are written, inspection of the underlying algorithms can help them learn methods for accomplishing the operations. Moreover, by programming extensions to the built-in facilities, students can reinforce understanding of the built-in and extended operations.

The above are ways that existing computer algebra systems can be used right now. However, there is a potential for much more. In conjunction with a computer-aided instruction package, existing computer algebra systems could be used for extremely flexible and intelligent algebra drill, testing, and tutorial dialogue.

None of the existing computer algebra systems is by itself a computer-aided math instruction system. The interactive muMATH lessons, for example, aim to teach muMATH rather than math. Moreover, these lessons use the built-in muSIMP driver in a very elementary "linear" presentation of the material. However, the muMATH driver can be replaced with any that the user desires, as explained in Section 13.14.1. Thus, someone experienced in computer-aided instruction could replace the muSIMP driver with one that was more appropriate for interactive lessons. Then computer-aided math lessons could be authored fully utilizing the mathematical capabilities of the system.

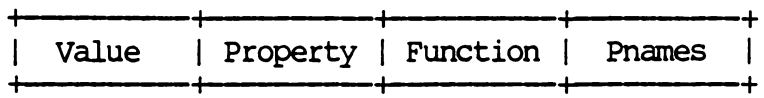
12. The muSIMP PROGRAMMING LANGUAGE

This section presents additional insights into the muSIMP programming language. It is intended as a reference for those who have already learned muSIMP through completion of the tutorial programming mode lessons of Section 16.

12.1 DATA STRUCTURES

muSIMP data is comprised of names, numbers, and nodes. Each type is recognizable and consists of a fixed number of **pointer cells** containing memory **addresses.** The cells can either point to other objects or to special-purpose entities outside the pointer space of objects. However, all three types have a **FIRST** cell and a **REST** cell. Moreover, these FIRST/REST cell pairs can only point to other objects within the pointer space. This eliminates the need for time-consuming **run-time type-checks** in the crucial selector functions that fetch these pointers.

12.1.1 Names



Internal Representation of a Name

A **name** is a recognizable data object consisting of four pointer cells. Names are uniquely stored in the sense that no two names in the system can have identical print names. Here are the uses of the four cells:

1. The FIRST or **value cell** contains a pointer to the name's current value, which is used by the evaluation functions. The value of a name is initialized to the name itself. This is known as automatic self-referencing or auto-quoting. The pointer is modified by the assignment functions or when a function is called which uses the name as a formal parameter in the function's definition.

2. The REST or **property list cell** contains a pointer to the name's property list, which is used by the property functions. Elements of this list are indicators adjoined to the corresponding property values. Property lists are initially set to the empty list.

3. The **function cell** contains a pointer to the name's function definition, if one exists. The contents of this cell cannot be accessed except by function applications, and the contents cannot be modified except by means of the function definition primitives. Until a name has been defined as a function, its function cell points to an undefined function trap.

4. The **Pname cell** contains a pointer to the string of ASCII characters used to print the name. This **print name string** can be of arbitrary

length. Access to this cell is restricted to the I/O and sub-atomic primitives. Print names are created or defined when a name is first used, and once defined, cannot be modified.

12.1.2 Numbers

Internal Representation of a Number

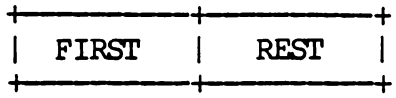
A **number** is a recognizable data object consisting of three pointer cells. Numbers are not uniquely stored, so duplicate numbers can coexist in storage. The cells are used as follows:

1. The FIRST cell contains a pointer to the number itself.

2. The REST or **sign cell** is initialized to FALSE if the number is non-negative; otherwise, it points to TRUE. The value of this cell is established when a number is created.

3. The **number vector cell** contains a pointer to the binary number vector which establishes the number's numerical value. It consists of a signed vector of up to 254 bytes. Thus, the magnitude of numbers is limited to 256^254, which is over 600 decimal digits.

12.1.3 Nodes



Internal Representation of a Node

Binary trees are the primary data structure in muSIMP. Internally they are implemented as a network of cell pairs called nodes. Each **node** consists of a FIRST cell and a REST cell. As mentioned earlier, the node's cells can only point to other bonafide muSIMP data objects: either a name, a number, or a node.

Nodes are often called **dotted-pairs**, because of their linearized external notation produced by PRINT or accepted by READLIST: The notation (X . Y) represents a node whose FIRST cell points to the object X, and whose REST cell points to the object Y.

Although the dot notation is more general, it is often more convenient to think of data as a linear list than as a deeply nested binary tree. For this purpose, **lists** are recursively defined as follows:

1. The empty list is denoted by the name FALSE.

2. If X is an object and Y is a list, then (X . Y) is a list.

A list of objects is printed by the function **PRINT** as a sequence of its elements separated by commas and delimited by parentheses. The function READLIST recognizes this notation for input. For example, if Y is the list (Y1, Y2, ..., Yn) then the dotted pair (X . Y) is printed as

(X, Y1, Y2, ..., Yn)

Conversely, the input of the form (X, Y1, Y2, ..., Yn) is recognized as (X . Y) by the **READLIST** function.

12.2 MEMORY MANAGEMENT

Dynamic, invisible **memory management** gives muSIMP much of its inherent power. This frees the programmer from having to bother with allocating suffient memory for a given problem. At any given time during the execution of a program, all of the memory not actually required to decribe the state of the machine should be available for any subsequent program use. This is achieved in muSIMP-80 in three phases.

12.2.1 Initial Data-space Partition

During the initialization phase of muSIMP, the amount of read/write memory available to the interpreter is first computed. Memory is then partitioned into four distinct data-spaces as given in the table below:

Ratio	Space	Contents
4:32	Atom Space	Name and number pointer cells
3:32	Vector Space	Print name strings and number vectors
23:32	Pointer Space	Nodes and D-code
2:32	Stack Space	Control/value stack

To create data objects required for running a program, space is taken from one of the above spaces. Space for a new name's or a new number's pointer cells is taken from the **atom space.** At the same time space for the associated print name string or number vector is taken from the **vector space.** The **pointer space,** which is by far the largest, provides storage for both D-code (i.e. the distilled code produced for compiled function definitions) and nodes. The combined control and value stack is located in the **stack space.** Based on our experience, these proportions approximate the relative use made of the spaces by most application programs.

12.2.2 Garbage Collection

New data structures are constantly being created during the execution of a muSIMP program, while others are implicitly discarded when they are no longer referenced. When the construction process uses up all available resources, a **garbage collection** is performed to reclaim the storage space vacated by discarded data structures, so that the user program can continue.

In muSIMP-80 the exhaustion of resources in either the atom, vector, or node spaces will cause collection to occur. First, those data structures accessible by means of chaining through pointer cells beginning either from a name cell or from a value stack entry are marked. Then during the second pass all the marked or active data objects are collected and compacted into one end of their respective data spaces, leaving the remainder of the spaces for new objects.

Although garbage collection is automatic, it is not entirely invisible to the user since it periodically causes a pause in the execution of a program. Less than half a second is required for the collection process in a 48K byte muSIMP-80 system using a 4MHz CPU clock. Normally this is of no concern to the programmer; however, it should be considered in the design of real time systems.

12.2.3 Reallocation of Data Space

If after a garbage collection occurs there is still insufficient free storage within a data space to continue program execution, the partitions of all the spaces are automatically moved to give more memory to the exhausted space. Thus muSIMP-80 can respond to changing demands placed on the various data spaces by differing application programs.

A phenomenon know as **thrashing** occurs when the system is forced to spend an inordinate amount of time garbage collecting for a very small number of nodes. The symptom of thrashing is a great increase in execution time for any given task. This problem can be resolved by increasing the computer's memory size or decreasing the amount of program and data storage requirements.

12.3 ERROR TRAPS AND DIAGNOSTICS

Errors in muSIMP are usually handled by informative diagnostic messages and the errant function returning an indication of the error. After the message is displayed, execution continues as before. It is the responsibility of the user program to recognize such errors and take appropriate action. The only error which is not handled in this way is the insufficient memory trap. It provides the option of either returning to the executive driver loop or going directly to the operating system.

12.3.1 Insufficient Memory Trap

Normally, automatically invoked garbage collections and dynamic reallocation of data spaces will provide sufficient storage in each space to continuously satisfy the demands of user programs. However, if the memory requirements for storing data objects finally exhaust all available resources, an **insufficient memory trap** will occur. The trap displays the following message on the console:

ALL Spaces Exhausted
Executive: ESC, ALT; System: Ctrl-C?

The user may then choose one of the options by typing the appropriate character. The "EXECUTIVE" option less drastic since it merely causes control to return to the muSIMP **executive driver** loop, without changing function definitions, property values, or name values from what they were just prior to the interrupt. The "SYSTEM" option terminates muSIMP, and returns control to the operating sytem.

12.3.2 Disk File I/O Errors

Disk errors may be caused by insufficient disk space or attempts to read past the end-of-file. The read and write disk error diagnostics are respectively:

End-Of-File Read
No Disk Space

12.3.3 Undefined Numerical Operations

If the second argument to any of the functions QUOTIENT, MOD, or DIVIDE is 0, a zero-divide trap occurs with the following diagnostic:

ZERO Divide Error

12.3.4 Input Syntax Error

Function PARSE can produce syntax error traps together with diagnostics of the following forms:

***** SYNTAX ERROR: expression USED AS NAME**
***** SYNTAX ERROR: expression USED AS PREFIX OPERATOR**
***** SYNTAX ERROR: expression USED AS INFIX OPERATOR**
***** SYNTAX ERROR: delimiter NOT FOUND**

where "expression" is the apparent offending portion of the input, and where "delimiter" is an apparently missing right delimiter such as a right parenthesis, ENDFUN, ENDSUB, EXIT, ENDLOOP, or ENDBLOCK. In any event, the remainder of the input from the point of confusion through the next terminator, such as ";", "$", or "&", is output to the terminal to help indicate the probable neighborhood of the cause. Examples which provoke the above four types are respectively:

```
5 (X);              {perhaps 5*(X) was intended?}

X Y;                {perhaps X*Y  was intended?}

X*/Y;               {perhaps X/Y was intended?}

WHEN ATOM(X, EXIT   {perhaps  WHEN ATOM(X) EXIT  was intended?}
```

12.4 IMPLEMENTING MACHINE LANGUAGE ROUTINES

Some specialized muSIMP applications may require the writing of machine language routines. For instance a user may wish to enhance muSIMP with graphics capability, or perhaps it is necessary to hand compile a particularly critical function for efficiency reasons.

Address typing is used by muSIMP to determine a function's type. This mandates that the beginning of all machine language subroutines be in low memory. A dummy jump table, beginning at location 0103H (hexadecimal), has been set aside for jumps from low memory to user defined routines. There is sufficient room for four (4) JMP instructions (i.e. opcode C3H). These jump instructions can be altered to jump to the address of the user defined routine wherever they are located. Depending on your system, additional room for jumps might be found in unused areas in page zero of memory.

Since muSIMP uses all available memory below the DOS (disk operating system), the best place to locate user defined routines is in the "protected" memory above the DOS. This mandates generating a DOS slightly smaller than what your computer is normally capable of supporting. Another alternative is to change the JMP instruction at location 0005H to an address less than its current address, thereby "fooling" muSIMP into believing it is operating under a smaller DOS. Of course another JMP instruction will have to be placed at the new address to jump to the DOS. This will free some memory just below the DOS for user routines.

All user defined machine language routines will be functions, not subroutines. They can have at most three arguments. If more than three arguments are required, they can be passed in as a list. The addresses of arguments are passed to machine language routines in the register pairs. The first argument is in the HL register, the second in the DE register, and the third in the BC register.

To return control to muSIMP, simply terminate all routines with a RET instruction. The returned value of the function will be the data structure pointed to by the HL register pair. It must be the address of a bona fide muSIMP data object (See Section 12.1). Even if no special value is desired to be returned, HL should still be set to some value such as FALSE. The value of FALSE can be determined by disassembling the machine language definition of the function EMPTY.

Linkage to machine language routines is done through the use of the muSIMP function PUTD. For instance, the following command will define FOO to be a machine-language function which begins at location 0103H:

```
PUTD ('FOO, 259);
```

The function GETD can be used to find the starting address of primitively defined muSIMP subroutines (See Section 13.9). Knowing these addresses, user defined subroutines can call the primitive subroutines directly.

13. muSIMP FUNCTIONS & OPERATORS

Every language must be described in terms of some language, which must be described in terms of some language, etc. Thus it is clear that at some point we must appeal to assumed inborn or culturally acquired understanding. However, the lengthy sequence of evasions can be avoided by using a partially circular description: muSIMP can be described in terms of muSIMP supplemented with English where necessary. Naturally, use of such a description requires some prerequisite partial knowledge of muSIMP gained by other means, just as use of an English dictionary requires some prerequisite knowledge of English. The interactive calculator-mode and programming-mode lessons described in Section 8 are intended to impart such a partial knowledge of muSIMP.

After one has initially learned the basics of muSIMP from the lessons, this self-descriptive type of reference information has the advantage of being compact and requiring no mastery of auxiliary notations. In addition, such documentation provides excellent nontrivial examples of structured programs written in muSIMP.

This section is a description of all of the primitively defined user-level functions, operators, control constructs, and control variables in muSIMP-80. For descriptive purposes only, we introduce some unindexed fictitious functions that are unavailable to the user. They are distinguished by having lower-case names. Lower-case is also used to distinguish English from legitimate muSIMP program constructs.

As a general rule, every function, operator expression, and control construct returns a value. Those which also modify the environment by assigning a value, modifying a property list, defining a function, or redirecting input or output are called **commands.** They are usually done for these effect more than their returned value. Commands are most often issued at the top-level of interaction, but they may also be imbedded in function definitions, to be executed when the task containing them is evaluated.

13.1 SELECTOR FUNCTIONS

Selector functions are used to select a desired sub-tree from a given binary tree. This gives a method for extracting information from the primary muSIMP data structure. **FIRST** and **REST** are the two fundamental selectors. Successive applications of these two functions is sufficient to traverse any tree. However, as indicated below, commonly-used compositions of FIRST and REST are defined in machine language for convenience and efficiency.

```
FUNCTION FIRST (X),
     The contents of the FIRST cell of X,
ENDFUN;
```

Remarks: The correct interpretation of FIRST(X) depends on whether X is an atom, and if not, whether it is regarded as a list or a binary tree: If X is an atom, then FIRST(X) returns the current **value** of X. If X is a list, then FIRST(X) returns the **first element** of that list. If X is regarded as a binary tree, then FIRST(X) returns the **left branch** of the tree.

```
FUNCTION REST (X),
    The contents of the REST cell of X,
ENDFUN;
```

Remarks: As with FIRST, the interpretation of REST(X) depends on whether X is an atom or a structure that is regarded as a list or a binary tree: If X is an atom, then REST(X) returns the **property list** of X. If X is a list, then REST(X) returns the **tail**, which is everything but the first element of that list. Finally, if X is a binary tree, then REST(X) returns the **right branch** of the tree.

```
FUNCTION SECOND (X),
    FIRST (REST (X)),
ENDFUN;
```

Remarks: SECOND(X) returns the second element of list X.

```
FUNCTION THIRD (X),
    FIRST (REST (REST (X))),
ENDFUN;
```

Remarks: THIRD(X) returns the third element of list X.

```
FUNCTION RREST (X),
    REST (REST (X)),
ENDFUN;
```

Remarks: RREST(X) returns the tail of the tail of X (i.e. the list consisting of all but the first two elements of X).

```
FUNCTION RRREST (X),
    REST (REST (REST (X))),
ENDFUN;
```

Remarks: RRREST(X) returns the tail of the tail of the tail of X (i.e. the list consisting of all but the first three elements of X).

13.2 CONSTRUCTOR FUNCTIONS

Constructor functions are used to generate data structures dynamically, during evaluation. In muSIMP such structures are usually a **list**, a **binary tree**, or a more general **directed graph**. These data structures are sufficiently general to model virtually any realizable data structure. The principal constructor named **ADJOIN** creates a new **node**. The storage required for this node is taken from the area of memory called **pointer space**. If previous ADJOINing has exhausted pointer space, then a garbage collection is automatically performed to reclaim the space used by no longer accessible data structures.

```
FUNCTION ADJOIN (X, Y),
     A new node whose FIRST cell is X and whose REST cell is Y,
ENDFUN;
```

Remarks: The correct interpretation of ADJOIN depends on how the data structures are regarded. When Y is a list, then ADJOIN(X,Y) returns the list whose first element is X and whose tail is Y. Otherwise, ADJOIN(X,Y) returns the tree whose left branch is X, and whose right branch is Y. Note that ADJOIN does not alter the structures X or Y.

```
SUBROUTINE LIST (X1, X2, ..., Xn),
     WHEN n = 0,  FALSE EXIT,
     ADJOIN (EVAL (X1), LIST (X2, X3, ..., Xn)),
ENDSUB;
```

Remarks: LIST takes an arbitrary number of arguments and returns a dynamically constructed list of the **evaluated** arguments. In contrast, X1 to Xn are not evaluated in a **constant** of the form

'(X1, X2, ..., Xn).

```
FUNCTION REVERSE (X, Y),
     WHEN ATOM (X),  Y EXIT,
     REVERSE (REST (X), ADJOIN (FIRST (X), Y)),
ENDFUN;
```

Remarks: REVERSE is normally used with only one list argument, in which case it returns a list having the same elements in the reverse order. However, when Y is also a list, then REVERSE(X,Y) returns a list whose elements are REVERSE(X) followed by those of Y.

```
FUNCTION OBLIST (),
     A list of the current primitive and user-introduced names,
ENDFUN;
```

Remarks: OBLIST() returns an **object list** containing all of the names currently in the system, ordered from the most recently introduced to the primitive names.

13.3 MODIFIER FUNCTIONS

Modifier functions redirect pointers in existing muSIMP data structures. Consequently, modifier functions are used more for their effect than their returned value. They can be used very effectively to modify already existing structures, thereby eliminating the need for the costly ADJOINing together of a whole new structure when only a minor part has been changed. However, the inexperienced muSIMP programmer should avoid these functions for the following reasons:

1. If an attempt is made to print a circular data structure created by use of these functions, endless printout will result.

2. Since data structures in muSIMP often share common sub-structures, the use of the modifier functions to change such a sub-structure will affect all the higher level data structures as well. This **side-effect** is often unforeseen and unwanted. Since the higher level structures are not necessarily named when they are changed, programs employing modifiers are hard to understand and debug.

There is no strong need to use these functions in most muSIMP application programs, including the muMATH approach to computer algebra. With these warnings, here then are the modifiers:

```
FUNCTION REPLACEF (X, Y),
     FIRST cell of X: Y,
     X,
ENDFUN;
```

Remarks: If X is a list, this function replaces the first element of X by Y. If X is a dotted-pair, it replaces the left element of X by Y. Finally, if X is an atom, it sets the value of X to Y.

```
FUNCTION REPLACER (X, Y),
     REST cell of X: Y,
     X,
ENDFUN;
```

Remarks: If X is a list, this function replaces the tail of X by Y. If X is a dotted-pair, it replaces the right element of X by Y. Finally, if X is an atom, it replaces the property list of X by Y.

```
FUNCTION CONCATEN (X, Y),
     WHEN ATOM (X),  Y  EXIT,
     WHEN ATOM (REST (X)),  REPLACER (X, Y)  EXIT,
     CONCATEN (REST (X), Y),
     X,
ENDFUN;
```

Remarks: This function concatenates, without using ADJOIN, the list X to the list Y. It accomplishes this by modifying the last tail pointer of X to point to Y. CONCATEN(X,X) will creates a circular list.

13.4 RECOGNIZER FUNCTIONS

Recognizer functions are used to identify data structures. They all take one argument and always return either TRUE or FALSE.

```
FUNCTION NAME (X),
     WHEN X is a name EXIT,
ENDFUN;
```

Remarks: NAME recognizes objects that are names.

```
FUNCTION INTEGER (X),
     WHEN X is an integer EXIT,
ENDFUN;
```

Remarks: INTEGER recognizes objects that are integers.

```
FUNCTION ATOM (X),
     NAME (X) OR INTEGER (X),
ENDFUN;
```

Remarks: ATOM recognizes objects that are atoms.

```
FUNCTION EMPTY (X),
     X = FALSE,
ENDFUN;
```

Remarks: EMPTY recognizes the empty list. Note that EMPTY has exactly the same definition as the logical NOT operator. Note that since the atom **FALSE** is used to denote the empty list, the recognizer function ATOM can also be used as a test for the end of a list. For example, see the definition of MEMBER in section 13.5.

```
FUNCTION POSITIVE (X),
     X > 0,
ENDFUN;
```

Remarks: POSITIVE recognizes positive integers.

```
FUNCTION NEGATIVE (X),
     X < 0,
ENDFUN;
```

Remarks: NEGATIVE recognizes negative integers.

```
FUNCTION ZERO (X),
     X = 0,
ENDFUN;
```

Remarks: ZERO recognizes the integer 0.

```
FUNCTION EVEN (X),
     ZERO (MOD (X, 2)),
ENDFUN;
```

Remarks: EVEN recognizes even integers.

13.5 COMPARATOR FUNCTIONS & OPERATORS

Comparator functions and **operators** are used to compare data structures. These predicates all require two arguments and return a value of either TRUE or FALSE.

```
PROPERTY RBP, EQ, 80;

PROPERTY LBP, EQ, 80;

FUNCTION EQ (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),  ZERO (X-Y)  EXIT,
     WHEN X and Y point to the same object  EXIT,
     FALSE,
ENDFUN;
```

Remarks: Normally the infix identity operator, EQ, is used for the identity comparision of atoms, i.e. names and numbers. However, in some circumstances it can be usefully used to compare non-atomic structures. For objects other than numbers, the function EQ returns TRUE if and only if its arguments are identical, i.e. they point to the same object or memory location. As described in Section 12, muSIMP names are uniquely stored. Thus EQ is an efficient test to determine the equality of names. However, since numbers are not stored uniquely, EQ has to and does compare the number vectors of numerical arguments.

```
PROPERTY RBP, =, 80;

PROPERTY LBP, =, 80;

FUNCTION = (X, Y),
     WHEN ATOM (X),  EQ (X, Y)  EXIT,
     WHEN ATOM (Y),  FALSE  EXIT,
     WHEN FIRST(X) = FIRST(Y),  REST(X) = REST(Y)  EXIT,
ENDFUN;
```

Remarks: The infix equality operator named "=" returns TRUE if and only if its arguments are equal. For atomic arguments, this is equivalent to the EQ test described above. However, the test is relaxed for non-atomic arguments in that they need only be **isomorphic** to one another. This means that they would look the same when printed. Since the EQ test is faster, it should be used in place of the "=" test in cases where at least one of the arguments is known to be atomic.

```
FUNCTION ORDERP (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),
          LESSP (X, Y)  EXIT,
     WHEN the address of object X is less that of object Y,
          TRUE  EXIT,
     FALSE,
ENDFUN;
```

Remarks: ORDERP provides a **generic ordering** for system names based on their order of introduction. Thus if the name X is introduced before the name Y (i.e. X occurs to the right of Y on OBLIST()) then ORDERP(X,Y) returns TRUE, otherwise returning FALSE. ORDERP has no obvious use for arguments that are not names.

```
FUNCTION ORDERED (X, Y),
     WHEN ATOM (Y),  ORDERP (X, Y) EXIT,
     WHEN ATOM (X),  TRUE EXIT,
     WHEN ORDERED (FIRST(X), FIRST(Y)),  TRUE EXIT,
     WHEN FIRST (X) = FIRST (Y),
          ORDERED (REST(X), REST(Y)) EXIT,
ENDFUN;
```

Remarks: ORDERED provides a simple yet efficient and useful **canonical** ordering for expressions, which can be used for applications such as sorting. For example, muMATH uses ORDERED to sort the terms of a sum or factors of a product. Thus, ORDERED could be replaced to give a different ordering of terms and factors.

```
FUNCTION MEMBER (X, Y),
     WHEN ATOM (Y),  FALSE  EXIT,
     WHEN X = FIRST(Y)  EXIT,
     MEMBER (X, REST (Y)),
ENDFUN;
```

Remarks: MEMBER (X, Y) returns TRUE if expression X is "=" to any member of the list Y, returning FALSE otherwise.

```
FUNCTION GREATER (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),
          X > Y  EXIT,
     FALSE,
ENDFUN;

FUNCTION LESSER (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),
          X < Y  EXIT,
     FALSE,
ENDFUN;
```

Remarks: These functions provide for greater than and less than comparisions of integers. Note that FALSE is returned if either argument is not an integer.

```
PROPERTY RBP, >, 80;

PROPERTY LBP, >, 80;
```

```
FUNCTION > (X, Y),
     GREATER (X, Y),
ENDFUN;



PROPERTY RBP, <, 80;

PROPERTY LBP, <, 80;

FUNCTION < (X, Y),
     LESSER (X, Y),
ENDFUN;
```

Remarks: In muSIMP these infix operators are entirely equivalent to the functions GREATER and LESSER respectively. However, the operators will be redefined by muMATH, whereas the functions remain defined only to compare integers.

13.6 LOGICAL OPERATORS

The **logical operators** permit Boolean combinations of **truth values.** As elsewhere in muSIMP, any non-FALSE value is considered to be logically TRUE.

```
PROPERTY RBP, NOT, 70;

FUNCTION NOT (X),
     X EQ FALSE,
ENDFUN;
```

Remarks: The prefix NOT operator returns TRUE if and only if its operand is FALSE. Thus NOT is equivalent to the function EMPTY.

```
PROPERTY RBP, AND, 60;

PROPERTY LBP, AND, 60;

SUBROUTINE AND (X1, X2, ..., Xn),
     WHEN n = 0, TRUE  EXIT,
     WHEN NOT EVAL (X1),  FALSE  EXIT,
     AND (X2, X3, ..., Xn),
ENDSUB;
```

Remarks: The AND infix operator returns TRUE if and only if each of its arguments evaluate to a non-FALSE value. Note that AND is a "subroutine" type of function, with its arguments sequentially evaluated until one evaluates to FALSE or until all have evaluated to non-FALSE values. Therefore, trailing arguments are not evaluated unnecessarily.

```
PROPERTY RBP, OR, 50;

PROPERTY LBP, OR, 50;

SUBROUTINE OR (X1, X2, ..., Xn),
     WHEN n = 0,  FALSE  EXIT,
     WHEN EVAL (X1),  TRUE  EXIT,
     OR (X2, X3, ..., Xn),
ENDSUB;
```

Remarks: The infix OR operator returns TRUE if any one of its arguments evaluates to a non-FALSE value. The arguments are successively evaluated, and if any evaluates to a non-FALSE value, TRUE is returned and none of the remaining arguments are evaluated.

13.7 ASSIGNMENTS

The modern trend toward an **applicative** style of structured programming encourages the use of recursion and functional composition, instead of assignment. Assignments can cause side effects which make programs more obscure, harder to debug, harder to verify, and harder to automatically exploit parallelism. As indicated in the programming mode lessons, it is quite feasible to program in muSIMP without ever making assignments but they are undeniably convenient for some purposes. Since the "=" sign is used for the equality comparison operator in muSIMP, the colon is used as the assignment operator. Also available is a less commonly used assignment function named ASSIGN which evaluates its arguments differently.

```
FUNCTION ASSIGN (X, Y),
    REPLACEF (X, Y),
    Y,
ENDFUN;
```

Remarks: This function sets the car or value cell of its first argument to that of its second, and returns the second. Note that the function is defined when the first argument is not a name, but its use in that situation is strongly discouraged. For more information, see the remarks following ":" operator.

```
PROPERTY RBP, :, 20;

PROPERTY LBP, :, 180;

SUBROUTINE : (X, Y),
    ASSIGN (X, EVAL (Y)),
ENDSUB;
```

Remarks: An expression of the form X: Y sets FIRST('X) to Y, returning Y as the value of the expression. Normally X is a name so that FIRST('X) is the value cell of the name X, yielding an assignment of the type prevalent in most traditional programming languages. However, X can be nonatomic, yielding side-effects similar to REPLACF but returning a different value. ":" is used far more often than the ASSIGN function because it is usually desired to assign a value to a variable rather than to the value of a variable. The distinction between the effect of ASSIGN and ":" on their arguments is demonstrated by the following example: If the value of DOG is FIDO then

```
DOG:  '(A DOBERMAN PINSCHER)
```

changes the value of DOG from FIDO to (A DOBERMAN PINSCHER). In contrast, if the value of DOG is FIDO then

```
ASSIGN (DOG, '(A DOBERMAN PINSCHER))
```

changes the value of **FIDO** to (A DOBERMAN PINSCHER), leaving the value of **DOG** unchanged as FIDO. One infrequent use of ASSIGN is to pass values back to the point of call of a subroutine, as described in Section 13.9.

```
SUBROUTINE POP (X),
     popl (X, EVAL (X))
ENDSUB;

     FUNCTION popl (X, Y),
          ASSIGN (X, REST (Y)),
          FIRST (Y),
     ENDFUN;
```

Remarks: If X is the name of a list, then POP(X) returns the FIRST of that list while setting X to the REST of the list. This operation is the muSIMP analog of the familar pop stack operation widely used in modern machine languages.

```
SUBROUTINE PUSH (X, Y),
     ASSIGN (Y, ADJOIN (EVAL (X), EVAL (Y))),
ENDSUB;
```

Remarks: If Y is the name of a list and X is an expression, then PUSH(X,Y) ADJOINs X onto the list Y, updating Y to point to this elongated list. This operation is the muSIMP equivalent of pushing information onto a stack.

13.8 PROPERTY FUNCTIONS

Property functions provide an excellent means of associating global properties with names. A name's **property list** is used to store these properties along with **indicator** tags. The property value associated with an indicator can be retrieved at any later time by using the GET function. These property lists facilitate construction of extremely flexible yet efficient data bases. For example, property lists are extensively used in muMATH in order to gain speed and modularity of the packages.

```
FUNCTION ASSOC (X, Y),
     WHEN ATOM (Y), Y EXIT,
     WHEN ATOM (FIRST (Y)), ASSOC (X, REST (Y)) EXIT,
     WHEN FIRST (FIRST (Y)) = X, FIRST (Y) EXIT,
     ASSOC (X, REST (Y)),
ENDFUN;
```

Remarks: ASSOC(X,Y) performs a linear search of the **association list** Y, looking for a non-atomic element whose FIRST is "=" to X. If found, the entire element is returned. Otherwise FALSE is returned.

```
FUNCTION PUT (X, Y, Z),
     WHEN EMPTY (GET (X, Y)),
          REPLACER (X, ADJOIN (ADJOIN (Y, Z), REST (X))),
          Z EXIT,
     REPLACER (ASSOC (Y, REST (X)), Z),
     Z,
ENDFUN;
```

Remarks: PUT(X,Y,Z) places on the property list of the name X under the indicator Y the property value Z, destroying any previous value under the same indicator.

```
SUBROUTINE PUTPROP (X, Y, Z),
     WHEN GET (X, Y) = Z OR GETD (GET (X, Y)) = Z, X EXIT,
     BLOCK
          WHEN NOT GET (X, Y) EXIT,
          WHEN SCAN EQ '$ EXIT,
          PRINT " *** REDEFINED:",
     ENDBLOCK,
     WHEN FIRST (Y) = 'FUNCTION,
          PUTD (PUT (X, Y, COMPRESS(LIST(X,Y))), Z),
          X EXIT,
     PUT (X, Y, Z),
     X,
ENDSUB;
```

Remarks: If Z is not already on the property list of the name X under the indicator Y, PUTPROP(X,Y,Z) places either Z or the compressed names X and Y on the property list of X under the indicator Y. If Z is a function body as described in Section 13.14, it is used to define the

function named COMPRESS(X Y). Making Z into a compiled function has the advantage of taking 1/3 the space and evaluating 20% faster. PUTPROP displays a warning message if the property value is being redefined.

```
FUNCTION PUTPROPER (EX1, EX2),
     WHEN NAME(EX1) AND NAME(EX2),
          LOOP
               WHEN NOT SCAN EQ COMMA EXIT,
               SCAN (),
          ENDLOOP,
          LIST (PUTPROP, EX1, EX2, PARSE(SCAN, 0)) EXIT,
     SYNTAX (),
ENDFUN;

PROPERTY PREFIX, PROPERTY,
     PUTPROPER (READLIST(SCAN), READLIST(SCAN));
```

The matchfix operator PROPERTY indicates a data-base command which can be used to place property values on a name. The following muSIMP command invokes the function PUTPROP to place the property Z under the indicator Y on the property list of X:

PROPERTY X, Y, Z;

If either X or Y is not a name, a syntax error occurs. If there is a previous value on X's property list under the indicator Y, it is deleted and a warning message is displayed. The three operands of a PROPERTY command are automatically quoted. Thus properties can be placed on operators without using the single quote mark.

```
FUNCTION GET (X, Y),
     X: ASSOC (Y, REST (X)),
     WHEN ATOM (X),  FALSE  EXIT,
     REST (X),
ENDFUN;
```

Remarks: GET(X,Y) returns the property value associated with the name X under the indicator Y if found; otherwise, FALSE is returned.

13.9 FUNCTION DEFINITION COMMANDS & FUNCTIONS

The **function definition** commands and functions are the only means of access to a name's function definition cell. As each function is defined, the definition is **pseudo-compiled** into an extremely dense form called **D-Code**, or **distilled code.** This results in about a 3 fold increase in code density and a 20% improvement in interpretation speed over a definition stored as a linked list. The inverse process of de-compiling a function definition back into a list also occurs automatically when GETD is used to retrieve a function definition. Thus this process of **incremental compilation** and de-compilation is invisible to the user, and the interactive nature of muSIMP is not compromised despite the dramatic improvement in efficiency and code density.

```
FUNCTION GETD (X),
     WHEN NOT NAME (X), FALSE EXIT,
     WHEN X is not a defined function or subroutine, FALSE EXIT,
     WHEN SUBR (X) OR FSUBR (X),
          the address of the machine-language routine EXIT,
     The list equivalent of the D-code defining the function X,
ENDFUN;
```

Remarks: GETD is used to get the definition of a muSIMP function or subroutine for further processing. If the function or subroutine is in machine language, then the physical memory address of the function is returned. Otherwise the list equivalent of the D-code is returned.

```
FUNCTION PUTD (X, Y),
     WHEN NOT NAME (X), FALSE EXIT,
     WHEN INTEGER (Y),
          Function cell of X: address given by Y,
          Y EXIT,
     Function cell of X: D-code equivalent of Y,
     Y,
 ENDFUN;
```

Remarks: If Y is an integer, then PUTD(X,Y) sets the function cell of X to the memory address equal to the number Y, modulo 64K. Otherwise the definition cell is set to the D-code equivalent of Y. The procedure for using PUTD to link to machine language subroutines is described in Section 12.4.

```
FUNCTION MOVD (X, Y),
     WHEN NOT NAME (X) OR NOT NAME (Y), FALSE EXIT,
     Function cell of Y: Function cell of X,
     GETD (Y),
ENDFUN;
```

Remarks: This function assigns the function cell of Y to point to the same memory location as that of the function cell of X. In cases where a MOVD is sufficient, it should be used instead of a GETD and PUTD, since no time or space is then wasted on a D-code compilation.

The **FUNCTION** command is used to define new muSIMP functions. The general form of the command is as follows:

```
FUNCTION name parameters,
     task1,
     task2,
      ...
     taskn
ENDFUN;
```

The function name can be omitted when there is no need to refer to it, such as when a nonrecursive function is stored on a property list using the PROPERTY command. If "parameters" is a name other than FALSE, the function can be called with an arbitrary number of arguments, which are passed to the function as a list assigned to the one parameter name. Otherwise, the parameters are bound to the corresponding argument values. Extra parameters beyond the number of arguments are initialized to FALSE, for use as **local variables.** Extra arguments beyond the number of parameters are evaluated but ignored.

Task evaluation within the function is performed successively until there are no more tasks or a non-FALSE predicate is evaluated. In the latter case evaluation proceeds as before except down the predicate's task list. In either case, the value of the function is the value of the last task evaluated. See Section 13.14 for details.

D-code does not accommodate dotted-pair constants within function definitions, so FALSE is used in place of any non-FALSE atomic REST cell in internal constants. Consequently, if a constant containing a dotted-pair is desired within a function definition, the user should assign the constant to a global variable outside the definition. The variable can then be used in place of the constant within the definition. For that matter, it is most efficient to use a similar technique even for list constants desired within function definitions.

The **SUBROUTINE** command is used to define muSIMP subroutines. The general form for the command is as follows:

```
SUBROUTINE name parameters,
     task1,
     task2,
      ...
     taskn
ENDSUB;
```

The arguments in a call to a subroutine are not evaluated before being passed to the subroutine. Otherwise, the evaluation of the subroutine is identical to evaluation of a function. As examples of the use of subroutines:

1. The AND and OR functions are defined in Section 13.6 as subroutines so that trailing arguments need not be evaluated unnecessarily.
2. The colon assignment operator is defined in Section 13.7 as a subroutine to prevent evaluation of the left argument.

13.10 SUB-ATOMIC FUNCTIONS

The **sub-atomic** functions are so named because they provide access to a name's print name string or a number's number vector. This makes it possible to temporarily unpack an atom's print name, operate on the resulting list of characters, and ultimately repack the list to form a new name. In addition to its sub-atomic capability the LENGTH function can be used to determine the top-level length of a list.

```
FUNCTION COMPRESS (X),
     WHEN ATOM (X), "" EXIT,
     WHEN NAME (FIRST (X)),
          Concatenate the print name of FIRST(X) onto the beginning
          of COMPRESS(REST(X)),
          return the corresponding name EXIT,
     WHEN INTEGER(FIRST(X)),
          Concatenate the print name string of the integer FIRST(X)
          with COMPRESS(REST(X)),
          return the resulting name EXIT,
     COMPRESS (REST (X)),
ENDFUN;
```

Remarks: COMPRESS returns a name whose print name is a packed version of the print names of the atomic members in list X. The current RADIX base is used to determine the print name string of numbers. Note that COMPRESS always returns a name, even if it only contains digits.

```
FUNCTION EXPLODE (X),
     WHEN NAME (X),
          Return a list of names whose print names correspond, in
          order, to the characters in the print name of X EXIT,
     WHEN INTEGER (X),
          Return a list of names whose print names are numerals
          corresponding to the digits of X expressed in the current
          radix base EXIT,
     FALSE,
ENDFUN;
```

Remarks: EXPLODE returns a list of names whose one-character print names correspond to the successive characters in the printed representation of atom X, returning FALSE if X is non-atomic. The current radix is used to determine the numerals representing a number. The numerals are then converted into the equivalent single character muSIMP names.

```
FUNCTION LENGTH (X),
     WHEN NAME (X),
          The number of characters in the print name of X EXIT,
     WHEN INTEGER (X),
          The number of digits required to print X EXIT,
     WHEN ATOM (REST (X)), 1 EXIT,
     1 + LENGTH (REST (X)),
ENDFUN;
```

Remarks: Although it also has the non-atomic application of returning the length of a list, the LENGTH function is categorized as sub-atomic because it determines the number of characters in the printed representation of atoms. Thus, LENGTH(X) is effectively three functions in one:

1. If X is a name, then the number of characters actually required to print X is returned. The current value of the control variable PRINT is taken into account while computing this length. The effect of the variable PRINT is described under the control variable subsection of Section 13.13.

2. If X is an integer, then the number of characters actually required to print X is returned. The current radix base and, if applicable, the leading "-" sign and/or leading "0" are properly taken into account while computing this length.

3. If X is a non-atomic, then the number of top-level nodes in the list is returned. As always in muSIMP, a REST which is atomic denotes the terminator of a list.

13.11 ARITHMETIC FUNCTIONS & OPERATORS

The arithmetic functions and operators implement exact integer arithmetic for numbers of magnitude up to 2^2032-1, which is greater than 600 decimal digits of accuracy. If given non-integer operands or if an **overflow** occurs, the functions and operators return a value of FALSE. Division by zero causes the following warning message to be displayed on the console:

ZERO Divide Error

and a value of FALSE is returned.

Although muSIMP does not provide rational arithmetic, the muMATH package named ARITH.MUS does generalize the primitive muSIMP operators "+", "-", "*", and "/" to treat rational operands, as described in Section 9.2. Consequently, muSIMP also provides primitive integer arithmetic functions named MINUS, PLUS, DIFFERENCE, TIMES and QUOTIENT so that integer arithmetic can be invoked directly if desired for extra efficiency when arguments are known to be integer.

```
FUNCTION MINUS (X),
     WHEN INTEGER (X),  -X  EXIT,
ENDFUN;


FUNCTION PLUS (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),  X + Y  EXIT,
ENDFUN;


FUNCTION DIFFERENCE (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),  X - Y  EXIT,
ENDFUN;


FUNCTION TIMES (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),  X * Y  EXIT,
ENDFUN;


FUNCTION QUOTIENT (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),
          WHEN Y = 0,  zero-divide error-trap  EXIT,
          WHEN POSITIVE (Y),  floor (X/Y)  EXIT,
          ceiling (X/Y)  EXIT,
ENDFUN;
```

Remarks: QUOTIENT returns a truncated integer quotient that is consistent with the MOD function defined below. This may not be what all users expect when X and/or Y is negative; however, the fundamental relation

$$X = Y*QUOTIENT(X,Y) + MOD(X,Y)$$

is always preserved by muSIMP.

```
FUNCTION MOD (X, Y),
     X - (Y * QUOTIENT(X,Y)),
ENDFUN;
```

Remarks: MOD(X,Y) is non-negative and periodic in X of period Y. This is the type of "remainder" that is most useful in number theory and computer algebra implementations, but it may differ from the notion of remainder that is held by some users and designers of some other programming languages, where consistent treatment of negative X and/or Y is rare.

```
FUNCTION DIVIDE (X, Y),
     WHEN INTEGER (X) AND INTEGER (Y),
          ADJOIN (QUOTIENT (X, Y), MOD (X, Y))  EXIT,
ENDFUN;
```

Remarks: DIVIDE returns a dotted pair consisting of the integer quotient and remainder of its arguments. This is much more efficient than computing them separately when when both results are desired. Note that dotted pairs do not display sensibly unless an expression is terminated with "&" or PRINT is used.

```
PROPERTY RBP, +, 100;

PROPERTY LBP, +, 100;

PROPERTY PREFIX, +, PARSE (SCAN, 130);

FUNCTION + (X, Y),
     PLUS (X, Y),
ENDFUN;
```

Remarks: The parser effectively ignores **unary prefix** "+", such as the second "+" in "3++5".

```
PROPERTY RBP, -, 100;

PROPERTY LBP, -, 100;

PROPERTY PREFIX, -, LIST ('-, PARSE(SCAN,130));

FUNCTION - (X, Y),
     WHEN EMPTY (Y),
          MINUS (X)  EXIT,
     DIFFERENCE (X, Y),
ENDFUN;
```

Remarks: "-" has a **unary prefix** role for designating negation, and "-" also has a **binary infix** role for designating subtraction. For example, 3- -5.

```
PROPERTY RBP, *, 120;

PROPERTY LBP, *, 120;

FUNCTION * (X, Y),
    TIMES (X, Y),
ENDFUN;


PROPERTY RBP, /, 120;

PROPERTY LBP, /, 120;

FUNCTION / (X, Y),
    QUOTIENT (X, Y),
ENDFUN;
```

Remarks: These two operators implement integer mulitplication and truncated division in muSIMP. They are extended by muMATH to handle rational arithmetic and utimately algebra.

13.12 READER FUNCTIONS & VARIABLES, AND PARSE FUNCTIONS

13.12.1 Reader Functions

The **reader** functions provide for character input to muSIMP programs. The functions read characters from the **current input source.** This input source can either be the console or any text file on the disk. The current input source can be controlled through the use of the function RDS in conjunction with the control variable RDS.

```
FUNCTION RDS (X, Y, Z),        % Input device ReaD Select %
     WHEN EMPTY (X),
          RDS: FALSE  EXIT,
     WHEN NAME (X) AND NAME (Y),
          WHEN EMPTY (Z),
               WHEN there exists a file named X.Y on the currently
               logged disk drive,
                    open X.Y for input,
                    RDS: X  EXIT,
               RDS: FALSE  EXIT,
          WHEN NAME (Z),
               WHEN there exists a file named X.Y on drive Z,
                    open X.Y on drive Z for input,
                    RDS: X  EXIT,
               RDS: FALSE  EXIT,
          RDS: FALSE  EXIT,
     RDS: FALSE  EXIT,
ENDFUN;
```

Remarks: The read select function is used to select an input source file. If the selected file is found, the file is opened for input, and the value of the variable RDS is set to the name of the file. Thus this file becomes the new current input source. If RDS is called with no arguments, with invalid arguments, or if the file is not found, the variable RDS is set to FALSE, making the console the current input source.

```
FUNCTION READCHAR (),
     Read one character from the current input source,
     SCAN: the corresponding muSIMP atom,
ENDFUN;
```

Remarks: This function reads and returns single characters from the current input source. Thus it necessitates that all token recognition be done by the application program. As an added programming convenience, the atom returned is also assigned to the muSIMP name SCAN. Integers are returned by READCHAR() only if the character is a decimal digit less than the current radix base.

FUNCTION **SCAN** (),
    Read one token from the current input source,
    SCAN: the corresponding name or number,
ENDFUN;

Remarks: This function reads tokens from the current input source and returns the corresponding muSIMP atom. A **token** is a string of characters delimited by either separator or break characters. Integers are delimited by the occurrence of a character which is not a digit in the current radix base. As an added programming convenience, the atom returned is also assigned to the muSIMP name SCAN.

Separator characters serve only to delimit tokens and are not returned by SCAN() as atoms. The SCAN() separator characters are:
space, carriage return, line-feed, and tab (CTRL-I).

Unlike separators, **break characters** are returned by SCAN() as single character muSIMP names. The SCAN() break characters are:

! $ & ') (* + , - . / @

; < = > ?] \ [^ _ { | } ~

Comments in an input source file can occur anywhere in the text so long as they are delimited by matching percent signs. The text of the comment is ignored by the function SCAN(). However, comments are echoed to the console on input of a source file as described in Section 13.12.2. The function READCHAR() reads and returns percent signs just as it would any other character.

Special characters such as the comment, separator, and break characters can be read in as names or parts of names by means of **quoted strings.** Such strings are delimited by double quote marks. The double quote itself can be included within the string by using two adjacent double quotes for each desired internal double quote.

If a disk file is the current input source and an attempt is made to read past the **end-of-file** (EOF) by either the function READCHAR() or SCAN(), the following warning message is displayed on the console:

End-Of-File Read

Also, the control variable RDS is set to FALSE, making the console the new current input source.

13.12.2 Reader Control Variables

RDS: FALSE;

Remarks: Normally control of the current input source is done through the use of the function RDS as described in Section 13.12.1. However, after a file has been opened and made current, control can be returned to the console without closing the input file, simply by setting the value of the variable RDS to FALSE within the file. A subsequent non-FALSE assignment to RDS made from the console will then return control to the point in the opened disk file at which reading was suspended. This technique is used to control the interactive lessons supplied with muSIMP/muMATH (see Section 8). Upon initial system startup and after interrupts or error traps, the muSIMP DRIVER function sets RDS to FALSE, making the console the current input source.

READ: 'READ;

Remarks: If the control variable READ is non-FALSE (i.e. the default value), lower-case letters are legitimate and distinct from their upper-case counterparts. The only exception to this is file names and types given to the functions RDS and WRS. They are always converted to upper case in order to eliminate conflicts with the operating system's file naming convention. If the READ is FALSE, then all lower case letters are converted to upper case as they are read in. However, lower case letters already in the system remain in lower case.

READCHAR: 'READCHAR;

Remarks: When the console is the current input source, the console input mode is controlled by the variable named READCHAR. Normally the value of the variable READCHAR is non-FALSE and console input is then in the **line-edit mode.** In this mode, when all the characters have been read from the current line, the operating system's line-edit routine is called for further input. Until a carriage return is typed, the system's normal editing procedures such as input echoing, backspacing, line deletion, printer output toggle using CTRL-P, etc. are in force. However, if READCHAR is FALSE, all buffering and input echoing is eliminated. This **raw input mode** is useful for immediate response to one character commands.

ECHO: FALSE;

Remarks: If a disk file is the current input source and the value of the control variable named **ECHO** is non-FALSE, then the characters being read from the file are echoed to the current output sink, which is usually the console. Note that since comments are also echoed, English language text within a comment can conveniently be displayed without having to actually process the text. This technique is used extensively in the interactive lessons supplied with muSIMP/muMATH. See Section 13.13.2 for other effects of ECHO.

13.12.3 Parse Functions

FUNCTION **PARSE** (X, Y),
From the current input source, read and parse the complete expression including the already-read token X, where Y is the right binding power of the operator to the left of X, if any, then return the resulting unevaluated object,
ENDFUN;

Remarks: **PARSE**(expr,rbp) is a function used to read a muMATH expression and convert it to list notation according to various rules established by operator's LBP and RBP binding powers and/or PREFIX or INFIX property rules as described below. When two operators compete for an operand between them, the operator with higher binding power towards the operand acquires the operand. In the case of a tie, the operator on the left acquires the operand.

Errors specific to PARSE include a member of DELIMITER "USED AS NAME", an infix operator "USED AS PREFIX OPERATOR", and a prefix or postfix operator "USED AS INFIX OPERATOR". The following properties are used by PARSE:

1. **INFIX** is a name on whose property list is stored expressions specifying how to parse **infix operators** for which mere left and right binding powers do not suffice. It is used for the assignment operator ":" since a check is made on its left operand to make sure it is a name. Also the INFIX property is used for "(" to correctly parse function calls written using mathematical notation. The respective operator's left operand is passed to the expression as the fluid name "EX1".

2. **PREFIX** is a name on whose property list is stored expressions specifying how to parse **prefix operators** for which mere left and right binding powers do not suffice. The **matchfix operators**, which include WHEN, LOOP, BLOCK, FUNCTION, SUBROUTINE, PROPERTY, and "(" when used to delimit a function's argument list, are examples of the use of the PREFIX property.

3. **LBP** is a name on whose property list is the integer **left binding powers** of infix and postfix operators. When two operators are competing for an operand, the operator with higher binding power toward the operand obtains the operand. In case of a tie, the left operator obtains the operand, so that infix operators with the same left and right binding powers associate left, as is usually desired.

4. **RBP** is a name on whose property list is the integer **right binding powers** of infix and prefix operators, for use as described for LBP.

```
RPAR:  '")";
LPAR:  '"(";
COMMA: '",";
```

Remarks: These constants should be used in place of parenthesis and/or commas, because the constants are free of any muSIMP parse properties.

```
DELIMITER: '(EXIT, ENDLOOP, ENDBLOCK, ENDFUN, ENDSUB,  ")",  ",");
```

Remarks: **DELIMITER** is a name that is initialized to the list of delimiters recognized by the muSIMP parser. When a matchfix operator is established, adjoining the matching delimiter to this list enables the parser to give more informative diagnostics by recognizing when a delimiter is used out of place. However, it is not necessary to adjoin delimiters to this list, and adjoining a delimiter has the effect of precluding its use out of context for other purposes.

```
FUNCTION TERMINATOR (),
     SCAN = ';  OR  SCAN = '$  OR  SCAN = '&,
ENDFUN;

FUNCTION DELIMITER (),
     TERMINATOR ()  OR  MEMBER (SCAN, DELIMITER),
ENDFUN;
```

Remarks: **DELIMITER()** is a predicate that returns FALSE if the current value of the name SCAN is neither a terminator nor on the list named DELIMITER.

```
FUNCTION MATCH (DELIM),
          % Uses the fluid variable SCAN set by SCAN () %
     WHEN SCAN = DELIM,  SCAN (),  FALSE  EXIT,
     WHEN SCAN = COMMA,  SCAN (),  MATCH (DELIM)  EXIT,
     WHEN DELIMITER (),  SYNTAX (DELIM, "NOT FOUND")  EXIT,
     ADJOIN (PARSE(SCAN,0), MATCH(DELIM)),
ENDFUN;
```

Remarks: **MATCH**(delim) is a function that parses zero or more expressions separated by commas and delimited by the value of its argument. MATCH returns a list of the parsed representations of these expressions.

```
PROPERTY PREFIX, WHEN,
     MATCH ('EXIT);

PROPERTY PREFIX, BLOCK,
     MATCH ('ENDBLOCK);

PROPERTY PREFIX, LOOP,
     ADJOIN ('LOOP, MATCH('ENDLOOP));
```

Remarks: The above commands define the parse properties for the WHEN-EXIT, BLOCK-ENDBLOCK, and LOOP-ENDLOOP control constructs. They all call MATCH() to find the construct terminator.

```
PROPERTY INFIX, "(", COND (
     WHEN NAME (EX1),
          ADJOIN (EX1, MATCH(RPAR)) EXIT,
     WHEN SYNTAX () EXIT);
```

Remarks: As shown by the above property, MATCH() is called by PARSE() when a left parenthesis is encountered in an infix position. This should only occur when a function application is being parsed. Since functions must be denoted by a name, not an integer or a list, a syntax error will occur if EX1 is not a name.

```
FUNCTION MATCHNOP (X, DELIM),
     WHEN SCAN EQ DELIM,  SCAN(),  X EXIT,
     SYNTAX (DELIM, "NOT FOUND"),
ENDFUN;

PROPERTY PREFIX, "(",
     MATCHNOP (PARSE (EX2, 0), RPAR);
```

Remarks: The function **MATCHNOP**(expr,delim) is used to verify that a matching "delim" was found following the PARSE of an expression. The only use made of it in muSIMP is to check that a matching right parenthesis was inserted following a left parenthesis.

```
FUNCTION READLIST (X),
     WHEN X EQ LPAR,
          LOOP
               WHEN NOT SCAN() EQ COMMA EXIT,
          ENDLOOP,
          WHEN SCAN EQ RPAR, SCAN (), FALSE EXIT,
          ADJOIN (READLIST(SCAN), READREST(SCAN)) EXIT,
     WHEN X EQ RPAR, SYNTAX () EXIT,
     SCAN (),
     WHEN X EQ COMMA, READLIST (SCAN) EXIT,
     X,
ENDFUN $

FUNCTION READREST (X),
     WHEN X EQ RPAR, SCAN (), FALSE EXIT,
     WHEN X EQ .,
          X: READLIST (SCAN()),
          WHEN SCAN EQ RPAR, SCAN (), X EXIT,
          SYNTAX () EXIT,
     ADJOIN (READLIST(X), READREST(SCAN)),
ENDFUN $

PROPERTY PREFIX, ', LIST (', READLIST(SCAN)) $
```

Remarks: The parse properties of the quote operator are given above; however, the definition of the quote function is given in Section 13.14.2. The operator really serves two purposes: a) as a function it supresses evaluation of its argument; b) as an operator it causes its operand to be read using list notation rather than the mathematical notation normally used in muSIMP.

```
FUNCTION SYNTAX X,
     WHEN ERR EXIT,
     ERR: TRUE,
     NEWLINE (),
     PRINT (" *** SYNTAX ERROR:  "),
     print each element in X,
     NEWLINE (),
     read and echo characters until a terminator is found,
     RDS: FALSE,
ENDFUN;
```

Remarks: SYNTAX exprs is a function that takes an arbitrary number of arguments. If the value of the global variable ERR is FALSE, then the message "*** SYNTAX ERROR: " is printed, followed by the arguments to SYNTAX separated by spaces. The remainder of the expression is read and printed but not parsed until a TERMINATOR character (i.e. a semi-colon, dollar sign, or and sign) is reached. Finally, the variable RDS is set to FALSE which returns control to the console.

muSIMP-80 OPERATOR BINDING POWER TABLE

Category	Operator	Operation	LBP	RBP
Ordering	(	Ordering	200	0
Assignment	:	Assignment	180	20
Numerical	!	Factorial	160	0
	^	Exponentiation	140	139
	*	Multiplication	120	120
	/	Division	120	120
	+	Addition	100	100
	-	Subtraction	100	100
	==	Equation	80	80
Comparison	=	Equality	80	80
	<	Less than	80	80
	>	Greater than	80	80
Logical	NOT	Negation	70	70
	AND	Conjunction	60	60
	OR	Disjunction	50	50

Note: When "+" and "-" are used as prefix operators a right binding power of 130 is used instead of 100.

13.13 PRINTER FUNCTIONS AND CONTROL VARIABLES

13.13.1 Printer Functions

The muSIMP **printer functions** direct character output to the **current output sink.** As determined by the function and the variable both named **WRS,** the sink can be either the console or a disk file.

```
FUNCTION WRS (X, Y, Z),                % Write Select %
     WHEN NOT EMPTY (WRS),
          Write out the final record of WRS and close the file,
          WRS: FALSE,
          WRS (X, Y, Z)  EXIT,
     WHEN EMPTY (X),
          WRS: FALSE  EXIT,
     WHEN NAME (X) AND NAME (Y),
          WHEN EMPTY (Z),
               On the currently logged in disk drive, if a file
                    named X.Y exists delete any existing file named
                    X.BAK, then rename the file X.Y to X.BAK, and
                    make a new directory entry for X.Y,
               WRS: X  EXIT,
          WHEN NAME (Z),
               On  drive Z, if a file named X.Y exists delete any
                    existing file named X.BAK, then rename the file
                    X.Y to X.BAK, and make a new directory entry
                    for X.Y,
               WRS: X  EXIT,
          WRS: FALSE  EXIT,
     WRS: FALSE,
ENDFUN;
```

Remarks: The **WR**ite **S**elect function is used both to select and later to close output sink disk files. If the current output sink is a disk file and WRS is called, that file is closed. Next, if a file name, type, and a drive (optional) are supplied as arguments to WRS, an already existing file of that name is renamed to a .BAK file of the same name. This provides an automatic one level **file backup** feature. Finally a new file of the given name and type is created for output on the appropriate drive, and the variable WRS is set to the name of the file. Thus this file becomes the new current output sink.

```
FUNCTION PRINT (X),
     WHEN NAME (X),
          Output to the current output sink the print name of X EXIT,
     WHEN INTEGER (X),
          Output to the current output sink the digits of X
               expressed in the current radix base, preceeded
               by a minus sign if X is negative  EXIT,
     PRINT (LPAR),
     prinlist (X), X,
ENDFUN;
```

```
        FUNCTION prinlist (X),
             PRINT (FIRST (X)),
             WHEN EMPTY (REST (X)),  PRINT (RPAR)  EXIT,
             SPACES (1),
             WHEN ATOM (REST (X)),
                  PRINT (". "),
                  PRINT (REST (X)),
                  PRINT (RPAR)  EXIT,
             prinlist (REST (X)),
        ENDFUN;

    FUNCTION PRINTLINE (X),
        PRINT (X),
        NEWLINE (),
        X,
    ENDFUN;
```

Remarks: PRINT(X) outputs X to the current output sink using list and dotted-pair notation. PRINTLINE(X) is identical, except that it also terminates the last line of output. If a disk file is the current output sink and there is insufficient disk space, the following warning message is displayed on the console:

No Disk Space

Then the control variable WRS is set to FALSe, making the console the current output file. Thus all subsequent output is sent to the console.

If the computer and operating system are appropriately configured for printer output and if console input is in the line edit mode as described in Section 13.12.2, then typing **CTRL-P** directs all muSIMP output displayed on the system console to the printer as well.

```
    FUNCTION LINELENGTH (X),
        WHEN  X > 11  AND  X < 256,
             Set maximum line-length to X,
             Return the previous line-length  EXIT,
        Return the current line-length,
    ENDFUN;
```

Remarks: If X is a number between 11 and 256, LINELENGTH (X) sets the length at which output lines will automatically be terminated to X, and the function returns the previous linelength. If X is not a number or is outside the permissible range, the current linelength is returned. The linelength is initially set to 72.

```
    FUNCTION RADIX (X),
        WHEN  X > 1  AND  X < 37,
             Set radix base to X,
             Return the old radix base  EXIT,
        Return the current radix base,
    ENDFUN;
```

Remarks: If X is a number from 2 to 36, then RADIX(X) sets the **radix base** in which numbers are expressed for both input and output, and the function returns the previous base expressed in the new radix base. If X is not a number or is outside the permissible range, the current radix base is returned. The radix base is initially set to ten.

```
FUNCTION NEWLINE (X),
    WHEN ZERO (X), FALSE EXIT,
    Output a carriage return and line feed to current output sink,
    WHEN POSITIVE (X) AND X < 256,
        NEWLINE (X-1) EXIT,
ENDFUN;
```

Remarks: If X is a non-negative number, then NEWLINE(X) outputs X number of new lines to the current output sink. Otherwise one new line is output to the sink.

```
FUNCTION SPACES (X),
    WHEN X > 0 AND X < 256,
        PRINT (" "),
        SPACES (X-1) EXIT,
    Return the current cursor position,
ENDFUN;
```

Remarks: If X is a non-negative number, SPACES (X) outputs X number of spaces to the current output sink. Otherwise no spaces are output to the sink. In any case, the resulting cursor position is returned.

```
FUNCTION PRTMATH (EX1, RBP, LBP, PRTSPACE),
    Taking account of declared binding powers and any special
    print rules on the property list of PRTMATH, print a deparsed
    representation of EX1, assuming
        a) the operator to its left, if any, has right binding
           power RBP, the operator to its right, if any, has
           left binding power LBP,
        b) appropriate spaces are to be printed if PRTSPACE is
           nonFALSE.
ENDFUN;
```

Remarks: PRTMATH(expr,rbp,lbp,prtspace) is a function which prints expr in standard mathematical form and encloses it within parentheses if the leading operator in expr has a left binding power less than or equal to RBP, or a right binding power less than LBP. Spaces are used around the top level operator if prtspace is non-FALSE. Thus, the normal top level usage of the function is

```
PRTMATH (expr, 0, 0, TRUE).
```

13.13.2 Printer Control Variables

WRS: FALSE;

Remarks: Normally control of the current output sink is done through the use of the function WRS, as described in Section 13.13.1. However, after a file has been opened for output, output can be directed to the console without closing the disk file simply by setting the value of the variable WRS to FALSE. A subsequent non-FALSE assignment to WRS will then redirect output to the disk file and append data onto the end of the file. Upon initial system startup and after interrupts or error traps, the default DRIVER function sets WRS to FALSE, making the console the current output sink.

PRINTLINE: 'PRINTLINE;

Remarks: The variable PRINTLINE controls the upper to **lower case conversion** of letters being output. Normally PRINTLINE is non-FALSE and all letters are printed as stored. However, if it is FALSE, all upper case letters are converted to lower case as they are output. This conversion in no way affects the internal storage of any name's print name.

PRINT: 'PRINT;

Remarks: If the variable PRINT is FALSE, names that contain separator or break characters are output using double quotes. This is necessary to permit the name to be subsequently read back in as the same name. Outputing such names using **quoted strings** is essential for such applications as muSIMP editors. Normally, however, PRINT is non-FALSE and a name's print name string is output without double quotes unless a double quote is actually part of the name.

ECHO: FALSE;

Remarks: If a disk file is the current output sink and the value of the control variable **ECHO** is non-FALSE, the characters being output to the file are also echoed to the console. See Section 13.12.2 for other effects of ECHO.

LPRINTER: FALSE;

Remarks: If the variable LPRINTER is non-FALSE, all muSIMP **output** to the current output sink, usually the console, will also be echoed to the line printer. Note that input, such as muSIMP commands typed in using the system console, will **not** be echoed to the printer although the resulting output will be. The usual way to produce hard copy listing of a muSIMP session is to use the CTRL-P option as described in section 4.6.1 of this manual. LPRINTER is useful primarily in muSIMP programs which need to direct output to the printer under program control.

13.14 DRIVER & EVALUATION FUNCTIONS AND CONTROL CONSTRUCTS

13.14.1 Driver Function

Controlling the interaction cycle is the job of the muSIMP function **DRIVER.** After performing some initializations, DRIVER enters an infinite loop which performs the following task: After printing the muSIMP prompt characters, the function PARSE is called to read input from the current input source and convert it to internal form. The function EVAL is then called to **evaluate** the expression. Finally the function PRTMATH is called to output the resulting value in mathematical notation to the current output sink.

```
FUNCTION DRIVER (
          % Local: % EX1, EX2),
     RDS: FALSE,
     WRS: FALSE,
     NEWLINE (2),
     LOOP
          ERR: FALSE,
          BLOCK
               WHEN ECHO (),
                    PRINT ("? "),
                    WHEN NOT RDS AND BELL,
                         PRINT ("^G")  EXIT EXIT,
          ENDBLOCK,
          EX1: FALSE,
          EX1: PARSE (SCAN(), 0),
          EX2: SCAN,
          BLOCK
               WHEN ECHO (),  NEWLINE (NEWLINE)  EXIT,
          ENDBLOCK,
          BLOCK
               WHEN ERR OR NOT TERMINATOR (),
                    SYNTAX (),
                    NEWLINE ()  EXIT,
               WHEN EX2 = '$,
                    @: EVAL (EX1),
                    WHEN ECHO (), NEWLINE () EXIT EXIT,
               PRINT ("@:"),
               @: EVAL (EX1),
               SPACES (1),
               BLOCK
                    WHEN EX2 = ';,
                         PRTMATH (@, 0, 0, TRUE) EXIT,
                    PRINT (@),
               ENDBLOCK,
               NEWLINE (2),
               NEWLINE (NEWLINE),
          ENDBLOCK,
     ENDLOOP,
ENDFUN;
```

BELL: TRUE;

Remarks: DRIVER is a function that controls the interaction cycle. After establishing the console as the current input and output file by setting both RDS and WRS to FALSE, the main read-evaluate-print "listen" loop is entered: An expression is first read by PARSE. If the terminator was the character ";", then the result is printed in mathematical notation by PRTMATH. In contrast, if the terminator was "&", then the result is printed in List notation. Finally, if the terminator was "$", then the result is not printed at all. However, in all cases it is assigned to the variable named @ unless an error occurred during the parse phase in which case an error message is displayed. For some applications, it may be desirable to (perhaps dynamically) replace this driver with another one or to recursively call DRIVER.

When the control variable **BELL** is non-FALSE, the ASCII bell character (control-G) is output to the console each time the question mark prompt is displayed. This is useful to signal the completion of a long computation if the console is not being continuously watched. If this feature becomes annoying, it can be turned off by setting the variable BELL to FALSE.

```
FUNCTION ECHO (),
     NOT RDS OR ECHO,
ENDFUN;
```

Remarks: The function ECHO() is a predicate that returns TRUE if input is being echoed to the terminal, returning FALSE otherwise.

13.14.2 Evaluation Functions

The evaluation functions are used for expression evaluation and program control. The algorithm for evaluating function bodies in muSIMP makes program control implicit in the structure of the body itself. This makes function definitions clean, short, and easy for the computer to interpret.

After a muSIMP function definition is parsed, the resulting linked list represents the definition. The first element of the list determines the function's type. It should be either the name FUNCTION or SUBROUTINE. FUNCTION indicates the function is a **call by value** (CBV) function. When a CBV function is called, arguments are first evaluated and only the resulting values are passed to the function. A function defined using SUBROUTINE is a **call by name** (CBN) function. A CBN function receives its arguments in unevaluated form, just as they were given in the call to the function. CBV functions are by far the most prevalent in muSIMP. Thus the novice programmer need not be concerned with learning to use the CBN functions until the other features of the language have been mastered.

The second element of a function's definition should be either a name or a list of names defining the function's **formal arguments.** If

the formal argument is a name which is not FALSE, the function is considered to be a **no-spread function.** A no-spread function receives its arguments as one list bound to the name. Thus no-spread functions can have an arbitrary number of arguments. However, if the second element is a list of atoms, the arguments will be passed to this **spread function** bound to each formal argument making up the list. Note that a function's being spread or no-spread is entirely independent of its being CBV or CBN.

The remaining elements of the list make up the definition's **function body.** The function body is a list of **tasks** which are to be successively performed when the function is called. The returned value of the function is the value of the last task performed. How a given task is to be performed depends on the structure of the task, as follows:

1. If the task is an atom, the value of the task is the value of the atom.

2. If the FIRST of the task is an atom, the first element is considered to be the name of a function which is to be applied to the list of arguments making up the REST of the task. The arguments are evaluated before the application for CBV functions.

3. If the FIRST of the FIRST of the task is an atom, the first element of the task is considered to be a **predicate,** which is then evaluated as described in 2) above. If the value of the predicate is FALSE, the value of the task is FALSE. However, if the predicate's value is non-FALSE, the original function body is abandoned and evaluation proceeds using the REST of the task as the new function body.

4. Otherwise the task is recursively evaluated as a function body itself before continuing with the evaluation of the top level function body. This permits conditional forks in function bodies to later recombine.

This evaluation scheme is very powerful but it does not have any provisions for non-recursive program control structures. Such **iterative** capability can be added to the algorithm above quite simply. A function body enclosed within the LOOP-ENDLOOP control construct will be evaluated as described above, except evaluation will start again at the beginning of the body instead of returning after the last task has been performed. This continues until a predicate as defined in case 3) above is non-FALSE. The value of the loop construct is the value of the function body following the non-FALSE predicate. Note that any number of predicates can occur within a loop at any desired location. This implementation of a LOOP preserves the basic evaluation method of muSIMP while greatly improving the performance of the language.

```
SUBROUTINE ' (X),
    X,
ENDSUB;
```

Remarks: The single-quote prefix operator has the effect of suppressing evaluation of its argument. For example, the value of a

variable named X can be cleared by the assignment X: 'X, and a file named LIM.SYS can be loaded from drive B by the command RDS('LIM,'B) even if the variable LIM and/or B have values other than themself.

The single-quote prefix operator also provides a mechanism to read data in the LISP-like dotted-pair and list notation rather than the functional and operator notation ordinarily presumed by the parser. See Section 13.12.3 for a discussion of the parsing effects of the single-quote.

```
FUNCTION EVAL (X),
     WHEN ATOM (X), FIRST (X) EXIT,
     WHEN NAME (FIRST (X)),
          WHEN undefined (FIRST (X)),
               WHEN FIRST (X) EQ EVAL (FIRST (X)),
                    evlis (X) EXIT,
               EVAL (ADJOIN(EVAL(FIRST(X)), REST(X))) EXIT,
          WHEN cbvp (GETD (FIRST (X))),
               APPLY (FIRST (X), evlis (REST (X))) EXIT,
          WHEN cbnp (GETD (FIRST (X))),
               APPLY (FIRST (X), REST (X)) EXIT,
          evlis (X) EXIT,
     WHEN cbvp (FIRST (X)),
          APPLY (FIRST (X), evlis (REST (X))) EXIT,
     WHEN cbnp (FIRST (X)),
          APPLY (FIRST (X), REST (X)) EXIT,
     evlis (X),
ENDFUN;
```

Remarks: EVAL(X) **evaluates** the expression X, with the help of the following auxiliary functions:

```
FUNCTION evlis (X),                  % Evaluate and return elements %
     WHEN ATOM (X), FALSE EXIT,                          % of list %
     ADJOIN (EVAL (FIRST (X)), evlis (REST (X))),
ENDFUN;

FUNCTION undefined (X),              % Undefined function recognizer %
     EMPTY (GETD(X)),
ENDFUN;

FUNCTION cbvp (X),                   % Call-by-value recognizer %
     machinefunction (X) OR Dcodefunction (X),
ENDFUN;

FUNCTION cbnp (X),                   % Call-by-name recognizer %
     machinesubroutine (X) OR Dcodesubroutine (X),
ENDFUN;

FUNCTION machinefunction (X),
     Return TRUE if X points to a machine-language call-by-value
     function, returning FALSE otherwise,
ENDFUN;
```

```
FUNCTION machinesubroutine (X),
     Return TRUE if X points to a machine-language call-by-name
     function, returning FALSE otherwise,
ENDFUN;

FUNCTION Dcodefunction (X),
     FIRST(X) EQ 'EXPR,
ENDFUN;

FUNCTION Dcodesubroutine (X),
     FIRST(X) EQ 'FEXPR,
ENDFUN;


FUNCTION APPLY (X, Y),
     WHEN NAME (X),
          WHEN undefined (X),
               WHEN X  EQ  EVAL(X),  FALSE  EXIT,
               APPLY (EVAL (X), Y)  EXIT,
          WHEN machinefunction (GETD (X)),
               WHEN ATOM (Y),
                    X (FALSE, FALSE, FALSE)  EXIT,
               WHEN ATOM (REST (Y)),
                    X (FIRST (Y), FALSE, FALSE)  EXIT,
               WHEN ATOM (RREST (Y)),
                    X (FIRST(X), SECOND(X), FALSE)  EXIT,
               X (FIRST (Y), SECOND (Y), THIRD (Y))  EXIT,
          WHEN machinesubroutine (GETD (X)),
               X (Y)  EXIT,
          WHEN Dcodefunction (GETD (X)) OR Dcodesubroutine (GETD (X)),
               bind (SECOND (GETD (X)), Y),
               Y: evalbody (FALSE, RREST (GETD (X))),
               Y  EXIT,
          FALSE  EXIT,
     WHEN Dcodefunction (X) OR Dcodesubroutine (X),
          bind (SECOND (X), Y),
          Y: evalbody (FALSE, RREST (X)),
          unbind (SECOND (X)),
          Y  EXIT,
     FALSE,
ENDFUN;
```

Remarks: APPLY(X,Y) applies the function X to the list of arguments Y. If X is a machine language routine, control passes to the routine. If X ia a function defined in D-code, the formal arguments of X are temporarily bound to the actual arguments, the function body is evaluated, the original values of the formal arguments are restored, and the value of the function body evaluation is returned. The following auxiliary functions are used by APPLY:

```
FUNCTION evalbody (X, Y),
     WHEN ATOM (Y),  X  EXIT,
     WHEN ATOM (FIRST (Y)) OR ATOM (FIRST (FIRST (Y))),
          evalbody (EVAL (FIRST (Y)), REST (Y))  EXIT,
```

```
        WHEN ATOM (FIRST (FIRST (FIRST (Y)))),
             X: EVAL (FIRST (FIRST (Y))),
             WHEN NOT X,
                  evalbody (FALSE, REST (Y))  EXIT,
             evalbody (X, REST (FIRST (Y)))  EXIT,
        evalbody (evalbody (X, FIRST (Y)), REST (Y)),
   ENDFUN;

   FUNCTION bind (X, Y),
        WHEN ATOM (Y),
             WHEN ATOM (X),  FALSE  EXIT,
             PUSH (EVAL (FIRST (X)), ARGSTACK),
             ASSIGN (FIRST (X), FALSE),
             bind (REST (X), Y)  EXIT,
        WHEN ATOM (X),  FALSE  EXIT,
        PUSH (EVAL (FIRST (X)), ARGSTACK),
        ASSIGN (FIRST (X), FIRST (Y)),
        bind (REST (X), REST (Y)),
   ENDFUN;

   FUNCTION unbind (X),
        WHEN ATOM (X),  FALSE  EXIT,
        unbind (REST (X)),
        ASSIGN (FIRST (X), POP (ARGSTACK)),
   ENDFUN;


SUBROUTINE LOOP (X1, X2, ..., Xn),
     evalloop (LIST (X1, X2, ..., Xn), LIST (X1, X2, ..., Xn)),
ENDSUB;

     FUNCTION evalloop (X, Y, Z),
          WHEN ATOM (Y),
               evalloop (X, X)  EXIT,
          WHEN ATOM (FIRST (Y)) OR ATOM (FIRST (FIRST (Y))),
               EVAL (FIRST (Y)),
               evalloop (X, REST (Y))  EXIT,
          WHEN ATOM (FIRST (FIRST (FIRST (Y)))),
               Z: EVAL (FIRST (FIRST (Y))),
               WHEN NOT Z,
                    evalloop (X, REST (Y))  EXIT,
               evalbody (Z, REST (FIRST (Y)))  EXIT,
          evalbody (FALSE, FIRST (Y)),
          evalloop (X, REST (Y)),
     ENDFUN;
```

Remarks: The LOOP function evaluates its argument in a manner identical to the evaluation of the clauses in a function body. However, if all the arguments are evaluated without a conditional having been satisfied, evaluation begins again with the first argument. The value of a LOOP construct is that of the last task evaluated.

```
SUBROUTINE COND (X1, X2, ..., Xn),
     evalcond (LIST (X1, X2, ..., Xn)),
```

```
ENDSUB;

    FUNCTION evalcond (X, Y),
        WHEN ATOM (X),  FALSE  EXIT,
        Y: EVAL (FIRST (FIRST (X))),
        WHEN NOT Y,
            evalcond (REST (X))  EXIT,
        evalbody (Y, REST (FIRST (X))),
    ENDFUN;
```

Remarks: COND successively evaluates the FIRST(X1), FIRST(X2), to FALSE. In the former case the REST of that argument is evaluated as a function body (see the interpretation of APPLY for details). In the latter case FALSE is returned by COND. COND is most useful outside a function definition when it is necessary to make a conditional statement.

13.14.3 Control Constructs

Control constructs are used within function definitions to indicate the desired structure of the definition. Although few in number, the muSIMP control consturcts result in an extremely flexible and natural programming style.

The **WHEN-EXIT** conditional control construct is used to evaluate predicates and take the appropriate fork in the program. This construct has the general form

```
WHEN predicate,
    body1  EXIT,
body2
```

Body2 begins immediately following the EXIT matching the WHEN and ends with one of the following statements: ENDFUN, ENDSUB, ENDLOOP, or ENDBLOCK. If the predicate evaluates to FALSE, then evaluation continues with body2. Otherwise, if the predicate is non-FALSE, body1 is evaluated and the result of that is the result of the entire clause.

The **LOOP-ENDLOOP** control construct can be used in function definitions to create an interative loop. It is implemented by involking the LOOP function and has the following form:

```
LOOP
    task1,
    task2,
    ......
    taskn,
ENDLOOP,
```

Each task is evaluated in turn until either a non-FALSE predicate in a WHEN-EXIT construct is encountered or all the tasks have been

evaluated. In the former case, the value of the WHEN-EXIT construct is the value of the LOOP. In the latter case, evaluation of the tasks begins again from the top. Since the LOOP construct parses to a function invocation, this construct can be used outside function definitions.

The **BLOCK-ENDBLOCK** control construct, like the WHEN-EXIT construct, is used to establish alternative forks in program control. However, upon termination of the fork, the function is **not** exited and control continues from the point immediately following the block. In muSIMP, the block construct has the following form:

```
BLOCK
   WHEN ... EXIT,
   task1,
   task2,
   ......
   taskn
ENDBLOCK,
```

As indicated, the first task within a block must be a WHEN-EXIT construct, and it can be followed by an arbitrary number of additional tasks. The special case in which there are no additional tasks is equivalent to the popular **if-then-else construct** found in many other computer languages. Since the remaining tasks within the block can be conditionals, the block provides all the control power of the **case construct** available in some computer languages. In addition to conditionals, tasks within a block can be any of the other muSIMP constructs including assignments, loops, or even another block. This results in an extremely flexible and powerful generalization of the more conventional case construct.

The evaluation of tasks within a block proceeds sequentially through the tasks unless a non-FALSE conditional causes a fork. When either the end of the block or the end of a fork is reached, control proceeds directly to the point following the block's ENDBLOCK delimiter. The value of a block is that of the last expression evaluated.

13.15 MEMORY MANAGEMENT FUNCTIONS

The **memory management function** is used to force a garbage collection and return the amount of currently available data space. Since memory management is fully automatic in muSIMP, normally there is no need to explicitly use the function except for its value. See Section 12.2 for a discussion of the memory management system.

```
FUNCTION RECLAIM (),
     Reclaim all un-referenced nodes by compacting all spaces,
     The amount of free space expressed in bytes,
ENDFUN;
```

Remarks: RECLAIM() forces a compacting type garbage collection to occur. It returns the total amount of available space in the pointer, name, and vector spaces expressed in bytes. Note that two bytes are required for each muSIMP pointer cell. Thus nodes require 4 bytes, numbers require 6 bytes, and names require 8 bytes. In addition names and numbers require storage for their respective print name strings and number vectors.

```
RECLAIM: 'RECLAIM;
```

Remarks: RECLAIM is also a control variable which controls printing of garbage collection statistics whenever a collection occurs. If RECLAIM is FALSE, the following is displayed on the console at the beginning and end of the garbage collection:

```
GC:  xxxx  yyyy  zzzz wwww
GC:  xxxx  yyyy  zzzz wwww
```

xxxx, yyyy, zzzz, and wwww represent 4 digit hexadecimal numbers which indicate the available space remaining in bytes of the atom, vector, pointer, and stack spaces respectively. See section 12.2 for a description of the meaning of these areas. The first line of the display indicates the sizes of the spaces before the collection, the second line indicates the sizes after the collection. This feature is primarily useful for determining which area is responsible for causing a thrashing problem.

13.16 ENVIRONMENT FUNCTIONS

The **environment functions** are used to save and load muSIMP **environments.** Prior to saving a system, all the data spaces are compacted into one area of memory so the resulting **SYS file** is of minimum size. The SYS file can later be loaded into a different size computer system than the one that produced the SYS file. Generally, after a muSIMP source file has been throughly debugged, a SYS file is created. A SYS file can be loaded much faster than reading in the equivalent source files every time.

```
FUNCTION SAVE (X, Y),
     WHEN NOT EMPTY (WRS),
          write out the final record of WRS and close the file,
          WRS: FALSE,
          SAVE (X, Y)  EXIT,
     WHEN NAME (X)  AND  NAME (Y),
          WHEN EMPTY (Y),
               Compact all current data structures,
               Save a memory image of the current environment
                    as a file named X.SYS on the current drive,
               TRUE  EXIT,
          Compact all current data structures,
          Save a memory image of the current environment as a
               file named X.SYS on drive Y,
          TRUE  EXIT,
     FALSE,
ENDFUN;
```

Remarks: This function saves the current muSIMP environment on a disk file. Since the currently active data structures are compacted before the save, the size of the "SYS" file increases with the amount of space occupied by function definitions and other data currently in the system.

```
FUNCTION LOAD (X, Y),
     WHEN NAME (X)  AND  NAME (Y),
          WHEN EMPTY (Y),
               Load X.SYS from the current disk,
               RDS: FALSE,
               Return directly to the executive DRIVER loop  EXIT,
          Load X.SYS from drive Y,
          RDS: FALSE,
          Return directly to the executive DRIVER loop  EXIT,
     FALSE,
ENDFUN;
```

Remarks: LOAD restores the muSIMP environment present at the time of the SAVE. LOAD does not return a value if the the "SYS" file is successfully loaded, but returns directly to the executive driver loop.

```
FUNCTION MEMORY (X, Y),
     WHEN X > -1  AND  X < 65536,
          WHEN Y > -1  AND  Y < 256,
               Replace the contents of memory location X with Y
               Return the original contents of X  EXIT,
          Return the contents of memory location X  EXIT,
     FALSE,
ENDFUN;
```

Remarks: This function permits any computer memory location to be examined or changed. If no second argument is given or if the second argument is out of range, only the value of the location is returned. **Warning:** The MEMORY function should be used only with great care since no protection whatsoever is provided against destroying the muSIMP interpreter or the computer's disk operating system.

```
FUNCTION SYSTEM (),
     Compact all data structures into low memory,
     Return to the operating system,
ENDFUN;
```

Remarks: When this function is executed, all data is compacted into low memory and then control is returned to the operating system. The compaction of data allows for a re-entry into muSIMP with the same environment that was present at the time of the call to the function SYSTEM. The re-entry address for CP/M versions of muSIMP is 100H (hexadecimal).

13.17 GRAPHICS and SPECIAL FUNCTIONS

A special implementation of muSIMP has been made for the **APPLE][microcomputer** running under **MicroSoft's Z80 SoftCard.** It has all the capabilities described in the earlier sections of this manual, plus additional primitively defined functions to take advantage of the computer's special hardware. In particular, **low-resolution graphics,** the **game control paddles,** and the **speaker** are supported in a manner very similar to BASIC.

```
FUNCTION TEXT (),
     Set  the screen to full-screen text mode (40 columns by 24
          lines),
     Move the cursor to the last line of the screen,
     FALSE,
ENDFUN;
```

Remarks: This function is used to return to the normal text mode following the use of the screen in the graphics mode. Like the BASIC version of the TEXT command, it does not clear the screen.

```
FUNCTION GRAPHICS (X, Y),
     WHEN EMPTY (X),
          Set  the screen for mixed mode (40 by 40), low-resolution
               graphics,
          WHEN 0 <= Y < 16,
               Fill the screen with the Y color number,
               Move the cursor to the four line text window,
               FALSE  EXIT,
          Clear the screen,
          Move the cursor to the four line text window,
          FALSE  EXIT,
     Set the screen for full-screen mode (40 by 48), low-resolution
          graphics,
     WHEN 0 <= Y < 16,
          Fill the screen with the Y color number,
          FALSE  EXIT,
     Clear the screen,
     FALSE,
ENDFUN;
```

Remarks: The GRAPHICS function is used to set up the screen for low-resolution graphics operations using the PLOT function. If its first argument is FALSE, a four line window at the bottom of the screen is left in the text mode as a programming convenience. Otherwise full-screen mode graphics is established. The second argument of the GRAPHICS function controls the color used to clear the screen. If it is not a valid color number, black is assumed.

Note: When this function is called with no arguments (e.g. GRAPHICS ();) it has the same effect as the APPLESOFT BASIC **GR** command.

```
FUNCTION PLOT (X, Y, Z),
     WHEN 0 <= Z < 16,
          WHEN 0 <= X < 40,
               WHEN 0 <= Y < 48,
                    Place a graphics dot of color number Z at
                         coordinates {X, Y},
                    TRUE  EXIT,
               WHEN 0 <= FIRST(Y) < 48  AND  0 <= SECOND(Y) < 48,
                    Draw a vertical line of color number Z at the
                         x-coordinate X, between the y-coordinates
                         FIRST(Y) and SECOND(Y),
                    TRUE  EXIT,
               FALSE  EXIT,
          WHEN 0 <= FIRST(X) < 40  AND  0 <= SECOND(X) < 40,
               WHEN 0 <= Y < 48,
                    Draw a horizontal line of color number Z at the
                         y-coordinate Y, between the x-coordinates
                         FIRST(X) and SECOND(X),
                    TRUE  EXIT,
               WHEN 0 <= FIRST(Y) < 48  AND  0 <= SECOND(Y) < 48,
                    Draw a rectangle of color number Z between the
                         x-coordinates FIRST(X) and SECOND(X), and
                         between the y-coordinates FIRST(Y) and
                         SECOND(Y),
                    TRUE  EXIT,
               FALSE  EXIT,
          FALSE  EXIT,
     WHEN 0 <= X < 40  AND  0 <= Y < 48,
          Return the color number of the graphics dot whose coor-
               dinates are {X, Y}  EXIT,
     FALSE,
ENDFUN;
```

Remarks: muSIMP conveniently combines in its single PLOT function the low-resolution graphics capability provided in APPLESOFT BASIC by the **COLOR, PLOT, HLIN, VLIN** and **SCRN** commands. In addition, rectangles of arbitrary dimension, location, and color can easily be painted.

The first two arguments of the PLOT function are used to specify the **x** and y coordinates of the 40 by 48 graphics screen. If these arguments are positive integers, they specify a single point on the screen. However, if either or both of them are lists consisting of two positive integers, they indicate a range of values for the respective coordinates. This range is then used for the plot and provides a means for drawing horizontal lines, vertical lines, and rectangles.

The optional third argument to PLOT is used to specify an APPLE][color. If no third argument is given or if it is not a valid color number, the PLOT function simply returns the current color number of the point specified by the first two arguments (i.e. {X, Y}). This provides the equivalent of the SCRN command in BASIC. However if it is a color number, the function will "plot" a graphics point of that color at those coordinates.

Note: The following is a list of the available colors and associated numbers:

0 black	4 dark green	8 brown	12 light green
1 magenta	5 grey 1	9 orange	13 yellow
2 dark blue	6 medium blue	10 grey 2	14 aquamarine
3 purple	7 light blue	11 pink	15 white

Example: The following function continuously plots points of given COLOR on the screen using the game paddles to determine the X and Y coordinates:

```
FUNCTION PLOTTER (COLOR),
     LOOP
          PLOT (QUOTIENT (PADDLE(0), 7),
                QUOTIENT (PADDLE(1), 6),
                COLOR),
     ENDLOOP,
ENDFUN;
```

```
FUNCTION PADDLE (X),
     WHEN 0 <= X < 4,
          Return the current value (a number between 0 and 255) of
               the game control paddle numbered X  EXIT,
     FALSE,
ENDFUN;
```

Remarks: This function is used to sense the current position of the game control paddles. The value returned is the same as that of the **PDL** function found in APPLE BASICs. The problem described in the APPLESOFT Manual of too rapidly reading different paddles has been resolved. Thus **no** artificial delay is required to obtain accurate readings.

```
FUNCTION BEEP (X, Y),
     WHEN 0 <= X < 256  AND  0 <= Y < 256,
          Creates a tone on the APPLE speaker of pitch X and
               duration Y,
          TRUE  EXIT,
     FALSE,
ENDFUN;
```

Remarks: The arguments to the BEEP function should be numbers between 0 and 255. The first argument specifies the pitch of the tone; 0 being the highest pitch, 255 being the lowest. The second argument specifies the duration of the tone; 0 being shortest, 255 being the longest (approximately a second). This function is intended for making sound effects only. No attempt has been made to match specific pitches with specific musical notes.

Example: The following loop produces a continues tone whose pitch and duration are controlled by the game paddle controls:

```
LOOP  BEEP (PADDLE(0), PADDLE(1))  ENDLOOP;
```

Because the APPLE][hardware is a known and predictable quantity, there are a few minor differences between the original CP/M version of muSIMP and the SoftCard version.

Instead of the value given in Section 13.13.1 of this manual, the default linelength is set to 39 for systems using the standard 40 column monitor. For systems with 80 column capability, the initial linelength remains 79. (The linelength is one less than the actual screen size to prevent unwanted blank lines on the screen resulting from the carriage return and linefeed automatically output by the APPLE monitor after a character is displayed in the rightmost column of the screen.)

Since all APPLEs have an **ESC** key and a delete (i.e. <--) key, the options available message given in Section 6.3 of this manual has been simplified to the following:

```
*** INTERRUPT:  To Continue Type: RET;
Executive: ESC   Restart: <--
System: Ctrl-C?
```

Naturally if your system only supports upper case characters, all lower case characters are converted to upper case on output.

Since the standard APPLE keyboard has no way of generating the left bracket (i.e. "["), the SoftCard version of CP/M automatically translates **CTRL-K** into the **left bracket.** This character is required for entering row vectors as described in Section 9.6. Similarly, SoftCard CP/M translates **CTRL-B** into the **back slash** (i.e. "\") character required for matrix division as described in Section 9.7.

As described in Chapter 5 of the **APPLE][Reference Manual,** the computer has a large number of "special memory locations" which are used to access the built-in I/O hardware. The muSIMP **MEMORY** function (see Section 13.16 of this manual) can be used to read or toggle these locations as required.

Since the SoftCard's **memory address translation** alters the locations of these I/O functions, the addresses given in the APPLE][manual for them are invalid. The following table lists some of the special memory locations and gives both their original 6502 address and the Z80 address in decimal and hexadecimal.

I/O Function	6502 Address Hex	6502 Address Decimal	Z80 Address Hex	Z80 Address Decimal
Keyboard Data Input	$C000	49152	$E000	57344
Clear Keyboard Strobe	$C010	49168	$E010	57360
Cassette Output	$C020	49184	$E020	57376
Speaker Toggle	$C030	49200	$E030	57392
Utility Strobe	$C040	49216	$E040	57408
Set GRAPHICS Mode	$C050	49232	$E050	57424
Set TEXT Mode	$C051	49233	$E051	57425
No Mix TEXT and GRAPHICS	$C052	49234	$E052	57426
Mix TEXT and GRAPHICS	$C053	49235	$E053	57427
Display Primary Page	$C054	49236	$E054	57428
Display Secondary Page	$C055	49237	$E055	57429
Display LO-RES graphics	$C056	49238	$E056	57430
Display HI-RES graphics	$C057	49239	$E057	57431
Annunciator Output #0 OFF	$C058	49240	$E058	57432
Annunciator Output #0 ON	$C059	49241	$E059	57433
Annunciator Output #1 OFF	$C05A	49242	$E05A	57434
Annunciator Output #1 ON	$C05B	49243	$E05B	57435
Annunciator Output #2 OFF	$C05C	49244	$E05C	57436
Annunciator Output #2 ON	$C05D	49245	$E05D	57437
Annunciator Output #3 OFF	$C05E	49246	$E05E	57438
Annunciator Output #3 ON	$C05F	49247	$E05F	57439
Cassette Input	$C060	49248	$E060	57440
Pushbutton Input #1	$C061	49249	$E061	57441
Pushbutton Input #2	$C062	49250	$E062	57442
Pushbutton Input #3	$C063	49251	$E063	57443
Game Control (Paddle) #1	$C064	49252	$E064	57444
Game Control (Paddle) #2	$C065	49253	$E065	57445
Game Control (Paddle) #3	$C066	49254	$E066	57446
Game Control (Paddle) #4	$C067	49255	$E067	57447

14. HOW TO LEARN MORE ABOUT ARTIFICIAL INTELLIGENCE

We expect that many who experience the muSIMP programming-mode lessons will be anxious to learn more about such programming techniques and how they are applied to artificial intelligence (AI) applications. Accordingly, here is a brief guide to the relevant professsional societies, literature, and programming languages.

14.1 THE PROFESSIONAL SOCIETIES

There are at least the following societies devoted to artificial intelligence or some aspect thereof:

1. **The American Association for Artifical Intelligence,** P.O. Box 3036, Stanford University, Stanford California 94305.

2. **The ACM Special Interest Group on Artificial Intelligence,** ACM, 1133 Avenue of the Americass, N.Y. N.Y. 10036.

14.2 AI PROGRAMMING LANGUAGES

LISP and LISP **surface languages,** such as muSIMP, are the languages most appropriate and most often used for artificial intelligence applications. For mainframe computers, the implementations of such languages are too numerous to list here. In contrast, such languages are only now becoming available for microcomputers. For example, The Soft Warehouse has developed a LISP implementation called muLISP for the same set of computers that run muSIMP, listed in Section 3. muLISP is generally available from the sources mentioned there.

14.3 THE LITERATURE

There is an extensive literature for artificial intelligence and LISP. Here are a few periodicals and books that serve as entry points into this literature:

Periodicals:

1. **Artificial Intelligence,** P.O. Box 211, 1000 AE Amsterdam, The Netherlands.

2. **Machine Intelligence,** Ellis Horwood Ltd., Market Cross House, Cooper St., Chichester, West Sussex, PO19 1EB England.

3. **IEEE Transactions on Pattern Analysis and Machine Intelligence,** Institute of Electrical and Electronics Engineers Inc., 345 East 47th St., N.Y., N.Y. 10017.

4. **Pattern Recognition**, Pergamon Press Inc., Maxwell Houssse, Fairview Park, Elmsford, N.Y. 10523.

5. **Robotics Age**, Box 801, La Canada, California 91011.

Books:

1. Allen, J. R., Anatomy of LISP,
McGraw-Hill Book Company, New York, NY, 1978.

2. Berkeley, E. C., and Bobrow, D. G., (eds),
The Programming Language LISP: Its Operation and Applications,
The M.I.T. Press, Cambridge, MA, 1964.

3. BYTE: The Small Systems Journal, LISP Issue, Vol 4, No 8,
BYTE Publications Inc., Peterborough, NH, August 1979.

4. Henderson, P., Function Programming: Application and Implementation,
Prentice-Hall, Englewood Cliffs, NJ, 1980.

5. Friedman, D. P., The Little LISPer,
Science Research Associates Inc.,
London, England, 1974.

6. McCarthy, J., Recursive Functions of Symbolic Expressions
and Their Computation by Machine,
Comm. ACM, pages 184-195, April 1960.

7. McCarthy, J., et al., LISP 1.5 Programmer's Manual,
The M.I.T. Press, Cambridge, MA, 1963.

8. Maurer, W. D., A Programmer's Introduction to LISP,
American Elsevier, New York, NY, 1973.

9. Siklossy, L., Let's Talk LISP,
Prentice-Hall, Englewood Cliffs, NJ, 1976.

10. Weissman, C., LISP 1.5 Primer,
Dickenson Publishing Co., Belmont, CA, 1968.

11. Winston, P. H., Artificial Intelligence,
Addison-Wesley Publishing Co., Reading, MA, 1977.

12. Winston, P. H. & Horn, B. K. P., LISP,
Addison-Wesley Publishing Co., Reading, MA, 1981.

LINELENGTH (78)$ #ECHO: ECHO$ ECHO: TRUE$

% If this lesson is being displayed too fast, it can be temporarily stopped by typing a CTRL-S (i.e. typing the letter "S" while depressing the CTRL key). Then type it again when you are ready to resume.

It is advisable to read sections 4 through 6 and 8 of the muMATH Reference Manual before beginning these lessons. This lesson can be aborted at any time by typing the ESCape key or the ALTmode key on your console followed by a CTRL-C.

In muMATH a "comment" is a percent sign followed by any number of other characters terminated by a matching percent sign. Thus, this explanation is a comment which has not yet been terminated. Comments do not cause computation; they are merely used to explain programs and examples to human readers. Here is an example of an actual computation%

```
1/2 + 1/6 ;
```
% Note how muMATH uses exact rational arithmetic, reducing fractions to lowest terms.

In muMATH, arithmetic expressions can be formed in the usual manner, using parentheses together with the operators "+", "-", "*", "/", and "^" respectively for addition, subtraction or negation, multiplication, division, and raising to a power. For example: %

```
(3*4 - 5) ^ 2 ;
```
% On some terminals, "^" looks like an upward-pointing arrow; on others it looks like a shallow upside-down letter V; and some terminals may employ an utterly different looking character which you may have to determine by experimentation.

The reason for using ^ and * is that standard terminals do not provide superscripts or centered dots or special multiplication crosses distinct from the letter X.

To prevent certain ambiguities, multiplication cannot be implied by mere juxtaposition. One of the most frequent mistakes of beginners is to omit asterisks.

Later, in order to give you an opportunity to try some examples, we will "assign" the value FALSE to the variable named RDS. When you are ready to resume the lesson, type the "assignment"

```
RDS: TRUE ;
```

including the semicolon and carriage return. This revises the value of the variable named RDS to the value TRUE. We will explain assignment in more detail later.

Don't forget that you can use local editing to correct mistypings on the current line. For example, on many operating systems, the key

marked RUBout or DELete cancels the last character typed on the line, and typing a CTRL-U cancels the current line. There is no way to modify a line after striking the RETurn key, but an expression can always be flushed by typing a final line containing a "grammatical" or "syntax" error such as "(;".

Now we are going to turn control over to you by setting RDS to FALSE. Try some examples of your own similar to the above. Also we suggest that you make a few intentional errors in order to become familiar with how they are treated. For example, try

5 7; 5+ /7; 5/0; and 0/0;

Have fun!: % RDS: FALSE ;

% The value resulting from the last input expression is automatically saved as the value of a variable named "@", which can be used in the next expression. For example: %

3 ;@ ^ @ ;@ ^ @;

% As this example illustrates, muMATH can treat very large numbers exactly and quickly. In fact, muMATH can accomodate numbers up to about 611 digits. To partially appreciate how large this is, compute the distance in feet or in meters to the star Alpha Centauri, which is 4 light years away, then use "@" to compute the distance in inches or in centimeters without starting all over. (In case you forgot, the speed of light is 186,000 miles/second or 300,000,000 meters/second.) %

RDS: FALSE ;

% Our answers are about 123,883,499,520,000,000 feet or 1,486,601,994,240,000,000 inches or 37,843,200,000,000,000 meters or 3,784,320,000,000,000,000 centimeters. Another dramatic comparison with 10^611 is that there are thought to be about 10^72 electrons in the entire universe. (Whoever counted them must be exhausted!)

Often one performs an intermediate computation or a trivial assignment for which there is no need to display the result. When this is the case, the display of the result can be suppressed by using a dollar sign rather than a semicolon as a terminator. For example, type

RDS: TRUE $

and note the difference from when you previously typed RDS:TRUE ; %

RDS: FALSE $

% It is often convenient to save values longer than "@" saves them, for use beyond the next input expression. The colon ASSIGNMENT operator provides a means of doing so. The name on the left side of the assignment operator is BOUND or SET to the value of the expression on its right. This value is saved as the value of the name until the name is bound subsequently to some other value. The name can be used as a variable in subsequent expressions, as we have used "@", in which case the name contributes its value to the expression. For example: %

RATE: 55 $ TIME: 2 $ DISTANCE: RATE * TIME ;

% Alphabetic characters include the letters A through Z, both upper and lower case, and the character "#". Note that the upper and lower case version of a letter are entirely distinct. Names can be any

sequence of alphabetic characters or digits, provided the first character is alphabetic. Thus X, #9, and ABC3 are valid names. Make an assignment of 3600 to a variable named SECPERHOUR, then use this variable to help compute the number of seconds in 1 day and 1 week: %
RDS: FALSE $
% Congratulations on completing CLES1.ARI. To execute the next lesson, merely enter the muMATH command

```
RDS (CLES2, ARI, drive);
```

where drive is the name of the drive on which that lesson is mounted. Alternatively, it may be advisable to repeat this lesson, perhaps another day, if this lesson was not perfectly clear. The use of any computer program tends to become much clearer the second time.

In order to experience the decisive learning reinforcement afforded by meaningful personal examples that are not arbitrarily contrived, we urge you to bring to subsequent lessons appropriate examples from textbooks, tables, articles, or elsewhere. Also, you are encouraged to experiment further with the techniques learned in this lesson: %

```
ECHO: #ECHO $
RDS () $
```

%File: CLES2.ARI (c) 08/10/80 The Soft Warehouse %

```
LINELENGTH (78)$  #ECHO: ECHO$  ECHO: TRUE$
```

% This file is the second of a sequence of interactive lessons on how to use the muMATH symbolic math system. This lesson presumes that the muMATH files through ARITH.MUS have been loaded.

For positive integer N, N factorial is the product of the first N integers. The "postfix" factorial operator is "!", which returns the factorial of its operand. For example, 3! yields 6, which is 1*2*3. Use this operator to determine the product of the first 100 integers: %

```
RDS: FALSE $
```

% The number base used for input and output is initially ten, but the RADIX function can be used to change it to any base from two through thirty-six. For example, to see what thirty looks like in base two: %

```
THIRTY: 30 $  RADIX (2) ;  THIRTY ;
```

% As you can see, the radix function returns the previous base, which is, of course, displayed in the new number base. This inforation helps to get back to a previous base. In base two, eight is written as 1000, so to see what thirty looks like in base eight: %

```
RADIX (1000) ;  THIRTY ;
```

% In base eight, sixteen is written as 20, so to see what thirty looks like in base sixteen: %

```
RADIX (20) ;  THIRTY ;
```

% As you can see, the letters A, B, ... are used to represent the digits ten, eleven, ... for bases exceeding ten. Now can you guess why we limit the base to thirty six?

In input expressions, integers beginning with a letter as the most significant digit must begin with a leading zero so as not to be interpreted as a name. For example, in base sixteen, ten is the letter-digit A, so to return to base ten: %

```
RADIX (0A) ;
```

% Why don't you now see what ninety-nine raised to the ninety-nine power looks like in base two and in base thirty-six, then return to base ten: %

```
RDS: FALSE $
```

% As you may have discovered, it is easy to become confused and have a hard time returning to base ten. Two is represented as 2 in any base exceeding 1, so a foolproof way to get from any base to any other is to first get to base two, then express the desired new base in base two. For example: %

```
RADIX (2) ;  RADIX (1010) ;
```

% Now we are guaranteeably in base ten, no matter how badly you got lost.

Now consider irrational arithmetic: Did you know that

```
(5 + 2*6^(1/2))^(1/2) -  2^(1/2) -  (3/2)^(1/2)
```

can be simplified to 0, provided we make certain reasonable choices of branches for the square roots? In general, simplification of arithmetic expressions containing fractional powers is quite difficult, but muMATH makes a valiant attempt. For example: %

```
4 ^ (1/2) ;  12 ^ (1/2) ;  1000 ^ (1/2) ;
```

% Try simplifying the square roots of increasingly large integers to gain a feel for how the computation time increases with the complexity of the input and answer: % RDS: FALSE $

% An input of the form (m/n)^(p/q) is treated in the usual manner as (m^(1/q))^p / (n^(1/q))^p . For example: %

```
(4/9) ^ (3/2) ;
```

% For geometrically similar people, surface area increases as the 2/3 power of the mass. Veronica wears a 1 square-meter bikini, and she is 50,653 grams, whereas her look-alike mother is 132,651 grams. Use muMATH to determine the area of her mother's similar bikini: % RDS: FALSE $

% 4^(1/2) could simplify to either -2 or +2, but muMATH picks the positive real branch if one exists. Otherwise, muMATH picks the negative real branch if one exists, as illustrated by the example: %

```
(-8) ^ (1/3) ;
```

% What if no real branch exits? Then muMATH uses the unbound variable named #I to represent the IMAGINARY number (-1)^(1/2), and expresses the answer in terms of #I, using the branch having smallest positive argument. For example: %

```
(-4) ^ (1/2) ;
```

% Decent simplification of expressions containing imaginary numbers, as described in lesson CLES4.ALG, requires that file ALGEBRA.ARI be loaded. Meanwhile if you believe in imaginary numbers and you can't contain your curiosity, why don't you experiment with them to see what muMATH knows about them: % RDS: FALSE $

% As with manual computation, picking a branch of a multiply-branched function is hazardous, so answers thereby obtained should be verified by substitution into the original problem or by physical reasoning. For this reason, there is a CONTROL VARIABLE named PBRCH, initially TRUE, which suppresses Picking a BRanCH if FALSE. For example: %

```
PBRCH: FALSE $  4 ^ (1/2);
```

% Users having a conservative temperament might prefer to do most of their computation with PBRCH FALSE.

This brings us to the end of CLES2.ARI. Though arithmetic, some of the features illustrated in this lesson may be foreign to you, because sometimes they are taught during algebra rather than before. Thus, if you have any algebra background whatsoever, we urge you to proceed to lesson CLES3.ALG even if some of CLES2.ARI was intimidating. Naturally, as implied by its type, file CLES3.ALG requires a muMATH system containing files through ALGEBRA.ARI.

If you decide not to proceed to algebra, but want to learn how to program using muSIMP, then proceed to lesson PLES1.ARI. %

```
ECHO: #ECHO$  PBRCH: TRUE$  RDS () $
```

```
LINELENGTH (78)$  #ECHO: ECHO$  ECHO: FALSE$
NUMNUM: DENNUM: 6$  DENDEN: 2$  NUMDEN: PWREXPD: 0$  PBRCH: TRUE$
X: 'X$  ECHO: TRUE$
```

% This file is the third of a sequence of interactive lessons on how to use muMATH in the calculator mode. This lesson presumes that the muMATH files through ALGEBRA.ARI have been loaded and that the user has studied the arithmetic lessons CLES1.ARI and CLES2.ARI.

An UNBOUND VARIABLE is one to which no value has been assigned. Mathematicians call such variables INDETERMINATES. You may have already inadvertently discovered that if you use an unbound variable in an expression, muMATH treats the variable as a legitimate algebraic unknown. Moreover, muMATH attempts to simplify expressions containing such unbound variables by collecting similar terms and similar factors, etc. For example: %

```
2*X - X^2/X ;
```
% See if muMATH automatically simplifies the expressions 0+Y, Y+0, 0*Y, Y*0, 1*Y, Y*1, Y^1, 1^Y, and 2*(X+Y) - 2*X. %
```
RDS: FALSE $
```
% Sometimes it is desirable to change a bound variable back to unbound status. This can be done by using the single-quote prefix operator, ', which looks like an apostrophe on many terminals. For example: %

```
EG: X + 5;  EG: 'EG;  EG + 2;
```
% Try assigning the value M*C^2 to E, then change E back to unbound status: % RDS: FALSE $

% You may have noticed that some of the more drastic transformations, such as expanding products or integer powers of sums, are not automatic. The reason is that such transformations are not always advantageous. They may make an expression much larger and less comprehensible. However, they may be necessary in order to permit cancellations which make an expression smaller and more comprehensible.

Accordingly, there are a few control variables whose values specify whether or not such transformations are performed. For example, the variable controlling expansion of integer powers of sums is called PWREXPD. This variable is conservatively initialized to zero, so that integer powers of sums are not expanded. For example: %

```
EG: (X+1)^2 - (X^2-2*X-1) ;
```
% Clearly this is an instance where expansion is desirable. When PWREXPD is a positive integer multiple of 2, then positive integer powers of sums are expanded, so let's try it: %

```
PWREXPD: 2 $  EG;
```
% Nothing happened!

The reason is that muMATH does not automatically reevaluate

previously evaluated expressions just because we change a control value. Not only would this be rather time consuming, but the ability to form expressions from other expressions constructed under different control settings provides a valuable flexibility for constructing partially expanded expressions.

On the other hand, it is often desirable to reevaluate expressions under the influence of new control settings, and the built-in EVAL function enables this: %

```
EVAL (EG) ;
%    Now that PWREXPD is 2, see how  (X+Y)^2 - (X-Y)^2 simplifies:  %
RDS: FALSE $
%    In muMATH, denominators are represented internally as negative
powers, and negative integer powers of sums are expanded if PWREXPD is a
positive integer multiple of 3.  For example:  %

PWREXPD: 3 $  1 / (X+1)^2 ;
%    What happens if  1 / ((X+1)^2 - X)^2  is evaluated under the influ-
ence of PWREXPD being 3?  For a little surprise, try it.% RDS: FALSE $
%    Even though  (X+1)^2 is WITHIN a negative power, it is itself a
positive power, so how about trying again with PWREXPD being 2*3:  %
RDS: FALSE $
%    Now, we would like to suggest a little experiment for you:  The
size limitation on algebraic expressions depends primarily upon the
amount of unemployed space available for storing the data structure used
to represent algebraic expressions.  We can always determine the total
amount of unemployed space expressed in bytes by the command:  %

RECLAIM ();
%    Numbers and nodes which are no longer a part of any value that we
can retrieve are automatically recycled intermittently, but the RECLAIM
function forces this "garbage collection" process.  The collection takes
on the order of a second, depending on memory size and processor speed;
and these slight pauses are sometimes noticable in the middle of a
printout or a trivial computation.  On a computer with front panel
lights, the collections are also usually recognizable by the change in
light patterns.
```

Naturally, if we load an extravagant number of muMATH files into a single muMATH dialogue or if we save a number of relatively large expressions as the values of variables, then there will be relatively little unemployed space for our next computation. Not only does this limit the size of the next expression, but computation time also increases dramatically as space becomes scarce, because relatively more time becomes devoted to increasingly frequent collections. The moral of the story is: don't unnecessarily load too many muMATH files or retain numerous expressions as the values of variables.

Now, for the experiment: In order to gain an appreciation for how computation time depends on the size of the input expression, answer, and unemployed storage, try timing each computation in the following sequence, until it appears that your space or patience is nearly exhausted:

```
     EG:(1+X)^2;  RECLAIM();  EG:EG^2;  RECLAIM();  EG:EG^2; ... %
RDS: FALSE $
%    These polynomials are called "dense", because there are no missing
terms less than the maximum degree in each unbound variable.  In
contrast "sparse" polynomials are missing a large percentage of the
possible terms less than the maximum degrees.  If you are still in an
experimental mood, you may wish to try the following analogous sequence
which produces extremely sparse results:
     RECLAIM(); (A+B)^2;  RECLAIM();  (A+B+C)^2;  RECLAIM(); ... %
RDS: FALSE $
%    Distribution of sums over sums is another transformation which can
dramatically increase expression size but is sometimes necessary to
permit cancellations.  For example, this transformation is clearly
desirable in the expression: %

EG: X^2 - 1 - (X+1)*(X-1) ;
%    When the control variable named NUMNUM is a positive integer
multiple of 2, then integers in NUMerators are distributed over sums in
NUMerators.  Similarly when the variable is a positive integer multiple
of 3, then monomials in numerators are distributed over sums in
numerators, whereas when the variable is a positive integer multiple of
5, then sums in numerators are distributed over sums in numerators.

     The reason for using the successive primes 2, 3, and 5, is that it
provides a convenient way to independently control the three types of
distribution using one easily remembered control variable name.

     The initial value of NUMNUM is 6, because numeric and monomial
distribution are recoverable (as we shall see), because neither
distribution dramatically increases expression size, and because a lack
of these distributions often prevents annoyingly obvious cancellations.
For instance the expression  2*(X+1) - 2*X  will not simplify unless
NUMNUM is a positive multiple of 2.  Similarly X+1 - (X+1)  will not
simplify to  0, since the expression is represented internally as
X+1 + -1*(X+1), which requires the -1 to be distributed over the sum.

     Thus, to return to our example, %

EG;  NUMNUM: 5 * NUMNUM;  EVAL(EG) ;
%    To witness the great variety of possible expansions, we set %

NUMNUM: 0 $  EG:  4 * X^3 * (1+X) * (1-X);
%    Now, successively EVAL EG with NUMNUM being 2, 3, 5, 6, 10, 15, and
30:  %  RDS: FALSE $
%    In interpreting these results, it is important to recall that
negations are represented internally as a product with the integer
coefficient -1, so NUMNUM must be a positive multiple of 2 to distribute
negations over sums.

     If positive values of NUMNUM cause expansion in numerators, how do
we request factoring in numerators?

     Negative values of NUMNUM cause factoring of numerators. Moreover,
the specific negative values cause factoring of the type which reverses
the corresponding expansion.  For example: %
```

```
X: 'X $  Y: 'Y $  NUMNUM: -2 $  EG: 10*X^2*Y + 15*X^3;
NUMNUM: 3*NUMNUM;  EVAL(EG);
```
% What about negative multiples of 5? Sorry folks, that's hard for computers as well as humans. However, we are working on it for future releases. Meanwhile, try out our semifactoring on the example

```
3*X*Y^3/7 - 15*X*Y^2/14 + 9*X^4*Y^2/7  % RDS: FALSE $
```

% As you may have guessed, there is a flag named DENDEN which controls expansion and factoring among negative powers in a manner entirely analogous to NUMNUM. Use it together with NUMNUM to expand the denominator then semifactor the denominator of the expression

```
X^2/((X-Y)*(X+Y) + Y^2 + X^2*Y)  % RDS: FALSE $
```

% You may have wondered why we chose the names NUMNUM and DENDEN. The reason is that there is another closely related control variable named DENNUM, which controls the distribution of various kinds of denominator factors over the terms of corresponding numerator factors:

> A positive multiple of 2 causes integer denominator factors to be distributed; a positive multiple of 3 causes monomial factors to be distributed; and a positive multiple of 5 causes sum factors to be distributed. For example: %

```
Y: 'Y $  DENNUM: DENDEN: NUMNUM: 0 $  EG: (5+3*X^2) / (15*X*(4+X));
DENNUM: 2 $  EVAL(EG);
DENNUM: 3*DENNUM;  EVAL(EG);
DENNUM: 5*DENNUM;  EVAL(EG);
```
% Positive setting of DENNUM and NUMNUM are particularly useful for work with truncated series or partial fraction expansions. For example, see if you can put the expression (6 + 6*X + 3*X^2 + X^3)/6 into a more attractive form: % RDS: FALSE $

% What about negative values of DENNUM?

A little reflection confirms that forming a common denominator reverses the effect of distributing a denominator. Thus, expressions are put over a common integer denominator when DENNUM is a negative integer multiple of 2, expressions are put over a common monomial denominator when DENNUM is a negative integer multiple of 3, and expressions are put over a common sum denominator when DENNUM is a negative integer multiple of 5. For example: %

```
X: 'X $  DENNUM: DENDEN: 0 $  EG:  1 + X/3 + (1+X)/X + (1-X)/(1+X);
DENNUM: -2 $  EG: EVAL(EG);
DENNUM: 3*DENNUM;  EG: EVAL(EG);
DENNUM: 5*DENNUM;  EG: EVAL(EG);
```
% Try fully simplifying the expression X^4/(X^3+X^2) + 1/(X+1) - 1 by expanding over a common denominator, then factoring: % RDS: FALSE $

% As with NUMNUM and DENDEN, the initial setting of DENNUM is 6, which causes distribution of numeric and monomial denominator factors over numerator sums. This tends to give attractive results for polynomials or series with rational-number coefficients, but the relatively costly common-denominator operation may be necessary for problems involving ratios of polynomials.

You have now been exposed to the four most important algebraic control variables in muMATH. Together with EVAL, the various

combinations of settings of these variables give rather fine control over the form of algebraic expressions. muMATH cannot read the user's mind, so it is important for the user to thoroughly master the use of these variables in order to achieve the desired effects.

Here are the most frequently useful combinations of settings for these three variables:

PWREXPD: 0; NUMNUM: DENDEN: DENNUM: 6; These initial values are usually good for general-purpose work, when the user wants to view some results before doing anything drastic or potentially quite time consuming.

PWREXPD: 6; NUMNUM: DENDEN: 30; DENNUM: -30; These settings yield a fully expanded numerator over a fully expanded common denominator. This form gives the maximum chance for combination of similar terms. Moreover, a rational function equivalent to 0 is guaranteed to simplify to 0. However, valuable factoring information may be irrecoverably lost.

PWREXPD: 0; NUMNUM: DENDEN: -6; DENNUM: -30; These settings yield a semifactored numerator over a semi-factored common denominator. This form gives the maximum chance for cancellation of factors between a numerator and denominator. However, the factoring is done incrementally, term by term, so it may be necessary to first expand over a common denominator so that all cancellable terms have an opportunity to cancel before attempting factorization.

PWREXPD: 2; NUMNUM: 30; DENDEN: -6; DENNUM: -30; These settings are a good compromise between the advantages of expansion and factoring. Semi-factoring is done in the denominator where it is usually most important, but there is a maximum opportunity for combination of similar terms in the numerator.

PWREXPD: 6; NUMNUM: DENDEN: DENNUM: 30; These settings are good for series expansions or partial fractions, because each term is fully expanded over its own denominator.

Again, we can't overemphasize the importance of mastering the use of these four control variables. They are your primary tool for imposing your will on the simplification process, and any lack of understanding of their proper use will ultimately lead to frustration. Accordingly, why don't you try the above and various other combinations on examples of your own choosing, until the usage becomes second nature: %
RDS: FALSE $
% Congratulations on completing CLES3.ALG. If the mathematical level was uncomfortably high, proceed to lesson PLES1.ARI. Otherwise proceed to CLES4.ALG. In either event, it is advisable to initiate a fresh muMATH environment, because our experiments have altered control values and made assignments which could interfere with those lessons in nefarious ways. %

ECHO: #ECHO$
RDS () $

```
LINELENGTH (78)$  #ECHO: ECHO$  ECHO: TRUE$
```

% This is the fourth of a sequence of muMATH calculator-mode lessons.

There are some other algebraic control variables besides PWREXPD, NUMNUM, DENDEN, and DENNUM; and they are occasionally crucial for achieving a desired effect. One of these, named NUMDEN, provides the logical completion of the latter three, by controlling the distribution of factors in numerators over the terms of denominator sums. NUMDEN is initially 0, but integer numerators are distributed over denominator sums when NUMDEN is a positive integer multiple of 2, monomial numerators are distributed over denominator sums when NUMDEN is a positive integer multiple of 3, and numerator sums are distributed over denominator sums when NUMDEN is a positive integer multiple of 5. For example: %

```
NUMNUM: DENDEN: DENNUM: 0 $  NUMDEN: 30 $
X / (X^3 + X + 1) / (Y + 1) ;  EG: (X+Y) / (1+X+Y) / (Y+1) ;
```

% Isn't that intriguing? It yields a sort of "continued-fraction" representation. Now for the reverse direction, which performs a denesting of denominators which is less drastic than a single common denominator: %

```
NUMDEN: -6 $  Z + 1 / (1/X + 1/Y) / (1+Y) ;
```

% See if you can devise examples exhibiting dramatic simplifications arising from either direction for this novel transformation. The fact that it so naturally complements NUMNUM, DENDEN, and DENNUM suggests that it must be useful for something! % RDS: FALSE $

% Another control variable named BASEXP controls distribution of a BASe over terms in an EXPonent which is a sum, or controls the reverse process which is collection of similar factors. As might be expected, integer bases are distributed over exponent sums when BASEXP is a positive integer multiple of 2, monomial bases are distributed over exponent sums when BASEXP is a positive integer multiple of 3, and base sums are distributed over exponent sums when BASEXP is a positive integer multiple of 5. Morever, the corresponding negative values cause collection of similar factors having the corresponding types of bases. BASEXP is initially -30. However, distribution (followed perhaps by collection) is sometimes necessary to let some of the terms in an exponent sum combine with the base. For example: %

```
EG: 2^(2+X) / 4 ;  BASEXP: 2 ;  EVAL (EG) ;
```

% See if you can devise an example which requires evaluating an expression first with sufficiently positive BASEXP, then reevaluating with sufficiently negative BASEXP, or vice-versa: % RDS: FALSE $

% Another control variable named EXPBAS controls the distribution of EXPonents over BASes which are PRODUCTS. Integer exponents are distributed over base products when EXPBAS is a positive integer multiple of 2, monomial exponents are distributed over base products when EXPBAS is a positive integer multiple of 3, and exponent sums are distributed over base products when EXPBAS is a positive integer multiple of 5. Naturally, the corresponding negative multiples request

collection of bases which have similar exponents of the indicated type. The initial value is 30, and here are some examples where distribution permits net simplification: %

```
(X^(1/2) * Y) ^ 2 ;  (X*Y)^2 - X^2*Y^2 ;  (4*X^2*Y) ^ (1/2) ;
```

% However, the user should beware that as with manual computation, distribution of noninteger exponents is not always valid. Consequently, conservative users may prefer to generally operate with EXPBAS being 2. Moreover, distribution of exponents tends to make expressions more bulky when no cancellations occur. For example %

```
(X * Y * Z) ^ (1/2) ;
```

% In fact, there are instances where negative settings of EXPBAS are necessary to acheive a desired result. For example: %

```
EG: 2^X * 3^X  + (1+X)^(1/2) * (1-X)^(1/2) - (1-X^2)^(1/2) ;
EXPBAS: -6 ;  NUMNUM: 30 ;  EVAL (EG) ;
```

% See if you can devise an example which requires evaluating an expression first with sufficiently positive EXPBAS, then reevaluating with sufficiently negative EXPBAS, or vice-versa, in order to simplify acceptably: % RDS: FALSE $

% The variable named PBRCH, already discussed in conjunction with fractional powers of numbers, also controls transformations of the form u^v^w --> u^(v*w). PBRCH is initially TRUE, but when PBRCH is FALSE, the transformation occurs only for integer w. Otherwise the transformation occurs for any w. The user should be aware that in some circumstances the selected branch is an inappropriate one, so that it may sometimes be necessary to set PBRCH to FALSE. See if you can devise such an instance: % RDS: FALSE $

% Now, try the examples 0^X and X^0, to see what happens: %

RDS: FALSE $

% The reason that 0^X is not automatically simplified to 0 is that 0^X is undefined for nonpositive values of X, so the transformation could lead to invalid results. Of course, sometimes users know that the exponent is positive, or they are willing to assume it is positive and verify the result afterwards. Consequently, there is a control variable named ZEROBAS, initially FALSE, which permits the transformation when nonFALSE.

Why then do we automatically simplify X^0 to 1 even though X could perhaps take on the value 0, giving the undefined form 0^0? Well, we also have a control variable for that, named ZEROEXP of course, but we initialized it to TRUE because:

1. If we are thinking of polynomials in X rather than any one specific value of X, then we are free to regard the polynomial X^0 as being formally equivalent to 1.

2. One cannot do effective simplification of rational functions without this widely accepted transformation.

3. Since 1 is the limit of X^0 as X approaches 0 from either side of the real axis, 1 is a reasonable interpretation even for 0^0.

Nevertheless, there is room for disagreement, and anyone who wishes is free to run with ZEROEXP FALSE. Why don't you try it, using some rational expression examples, in order to see how you feel about this issue? % RDS: FALSE $
% It is easy to forget the current control-variable settings, and it is even easy to forget the existence of certain control-variables, so we have provided a handy-dandy function named FLAGS which returns the empty name "" after printing a display of all the flags and their values: %

```
FLAGS ();
```

% If you ever get frustrated because you can't get an answer close to the desired form, try this command. It may reveal some inappropriate settings or remind you of some alternatives you forgot, or reveal the existence of potentially relevant flags of which you were unaware.

Often a dialogue proceeds best under some control settings which are suitable for the majority of the computations, with an occasional need for an evaluation under different control settings. Each such exception could involve new assignments to several control variables, followed by an evaluation then assignments to restore the variables to their usual values. This process can become tedious, and baffling effects can result from inadvertently forgetting to restore a control variable to its usual value. Consequently, as a convenience, we have provided some functions which, for the most commonly desired sets of "drastic" control values, establishes these values, reevaluates its argument, then allows the control variables to revert to their former values before returning the reevaluated argument.

One of these functions is called EXPAND, because it requests full expansion with fully distributed denominators, bases, and exponents. More specifically, it uses the following settings:

```
PWREXPD: 6; NUMDEN: 0; NUMNUM: DENDEN: DENNUM: BASEXP: EXPBAS: 30;
```

To see its effect, try EXPAND (((1+X)/(1-X))^2); % RDS: FALSE $
% Another one of these convenience functions is called EXPD, and it fully expands over a common denominator. Thus the internal control settings are the same as for EXPAND, except that DENNUM: -30. Try

```
EXPD (1/(X+1) + (X+1)^2);
```

% RDS: FALSE $
% Finally, there is a convenience function named FCTR, and it semi-factors over a common denominator. It evaluates its argument under the following control-variable settings:

```
NUMNUM: DENDEN: -6;  DENNUM: BASEXP: EXPBAS: -30;  PWREXPD: NUMDEN: 0;
```

Since semi-factoring is done termwise, it may be necessary to use EXPD before applying FCTR to an expression, in order to get the desired result. See if you can devise an instance where this is true: %
RDS: FALSE $
% This brings us to the end of lesson CLES4.ALG. The next lesson is CLES5.ALG, but as before, it is advisable to start a fresh muMATH to avoid conflicts with bindings established in the current lesson. %

```
ECHO: #ECHO $
RDS () $
```

```
LINELENGTH (78)$  #ECHO: ECHO$  ECHO: TRUE$
```

% It is often desired to extract parts of an expression. Particularly frequent is a need to extract the numerator or denominator of an expression. Accordingly, there are built-in SELECTOR functions named NUM and DEN for this purpose: %

```
DENNUM: 0 $  EG: (1+X) / X ;  NUM (EG) ;  DEN (EG) ;
NUM (1 + EG);  DEN (1 + EG);
```

% As the last two examples illustrate, NUM and DEN do not force a common denominator or any other transformation before selection, so the denominator is always 1 when the expression is a sum or when the expression is a product having no negative powers. Try out NUM and DEN on a few examples of your own to gain some experience: % RDS: FALSE $
% The Programming-mode lessons will explain how to completely dismantle an expression to get at any desired part, such as a specific term, coefficient, base, or exponent.

muMATH represents the imaginary number (-1)^(1/2) as #I, and muMATH does appropriate simplification of integer powers of #I. For example: %

```
#I ^ 7 ;  EXPAND ((3 + #I) * (1 + 2*#I)) ;  EXPAND ((X + #I*Y) ^ 3) ;
```

% Try it, you'll like it! % RDS: FALSE $
% The definition of the operator "^" in file ALGEBRA.ARI also implements two higher-level transformations which we mention here only in passing:

muMATH represents the base of the natural logarithms as #E and the ratio of the circumference to the diameter of a circle as #PI. Using these, muMATH performs the simplification

$$\#E \text{ ^ } (n * \#I * \#PI / 2) \longrightarrow \#I\text{^}n,$$

where n is any integer constant, after which the power of #I is reduced appropriately. Also, if a control variable called TRGEXPD is a negative multiple of 7, then complex exponentials are converted to trigonometric equivalents. (The opposite transformation for sines and cosines to complex exponentials for TRGEXPD = 7, is implemented by file TRGPOS.ALG.) If your mathematical background includes these facts, you might wish to experience them here. Otherwise you can safely ignore this digression: % RDS: FALSE $
% You may have wondered whether or not an assignment to a variable, say X, automatically updates the value of a bound variable, say EG, which was previously assigned an expression containing X. Let's see: %

```
X: 5 $  Y: 'Y $  EG: X + Y ;  X: 3 ;  EG;  EVAL (EG) ;
```

% Apparently the answer is "no", at least if X is bound when the assignment to EG is made. This should not be surprising, because after contributing its value to the expression X + Y, all traces of the name X are absent from this expression. However, suppose that we do a similar calculation wherein X is initially unbound: %

```
X: 'X $  EG: X + Y;  X: 3;  EG;
```
% As when we change control variables, previously evaluated expressions are not automatically reevaluated when we bind an unbound varible therein. However, we can always use EVAL to force such a reevaluation: %

```
EVAL (EG) ;
```
% Since we did not assign the result to EG, reevaluation of EG after a different assignment to X still has an effect: %

```
X: 7 $  EG: EVAL (EG);
```
% Since this time we did assign the result to EG, further changes to X can have no effect on EG regardless of evaluation: %

```
X: 9 $  EG: EVAL (EG) ;
```
% If these examples are not entirely clear, you had better take the time to experimentally learn the principles by trying some examples of your own: % RDS: FALSE $
% It is often desired to reevaluate an expression under the influence of a temporary local assignment to one of the variables therein without disturbing either the existing value of the variable or else its unbound status. The built-in EVSUB function provides a convenient method of accomplishing this effect. EVSUB returns a reevaluated copy of its first argument, wherein every instance of its second argument is replaced by its third argument. For example: %

```
NUMNUM: 6 $  M: 'M $  C: 'C $  V: 'V $  EG: M*C^2 + M*V^2/2 $
EVSUB (EG, M, 5);  EVSUB (EG, M, M1+M2);  M;
```
% Play around with EVSUB for awhile until you are absolutely sure that you understand the difference between substitution and assignment: %
```
RDS: FALSE $
```
% You may have discovered that EVSUB also permits substitution for arbitrary subexpressions as its second argument. For example: %

```
M: 'M $  C: 'C $  E: 'E $  EVSUB (M*C^2 + 7, M*C^2, E);
```
% To keep the algebra package small, we have not endowed EVSUB with any sophistication about finding algebraically IMPLICIT instances of its second argument in its first. See if you can find examples where EVSUB does not do a substitution that you would like it to do: % RDS: FALSE $
% Here is an example where a desired substitution doesn't fully occur:%

```
NUMNUM: 6 $  C: 'C $  S: 'S $  EVSUB (1 - 2*S^2 + S^4,  S^2,  1 - C^2);
```
% The reason we did not get the desired simplification to C^4 is that if the second argument is a power, it matches only the same power in the first argument. We can usually circumvent such problems by instead using an equivalent substitution wherein the second argument is a name rather than a power. For example: %

```
PWREXPD: 2 $  EVSUB (1 - 2*S^2 + S^4,  S,  (1-C^2) ^ (1/2));
```
% Here is a somewhat different example wherein a desired substitution does not occur: %

```
EVSUB (2*C*S, C*S, C2);
```
% The reason is that if the second argument is a product, it matches only the same COMPLETE product in the first argument. Again, the remedy

is to use an equivalent substitution wherein the second argument is a name. For example: %

```
EVSUB (2*C*S, C, C2/S);
```
% Here is a final example for which a desired substitution does not occur: %

```
EVSUB (C^2 + S^2 - 1 + C + S,  C^2 + S^2,  1);
```
% Similarly to products, if the second argument is a sum, it matches only the same COMPLETE sum in the first argument. As before, we could circumvent the difficulty by making an equivalent substitution of (1-C^2) ^ (1/2) for S, or (1-S^2) ^ (1/2) for C, but that would leave an ugly square root in the answer. If our goal is to delete the subexpression C^2 + S^2 - 1, then we can use to our advantage the fact that powers must match exactly for a substitution to take place: %

```
EVSUB (C^2 + S^2 - 1 + C + S,  C^2,  1 - S^2) ;
```
% See now if you can use such techniques to get your examples to work:
% RDS: FALSE $
% This brings us to the end of the calculator-mode lessons. There are, of course, higher-level math packages in muMATH, but the fact is that from a usage standpoint, we have already covered the hardest part, which is understanding evaluation, substitution, and the ramifications of the various algebraic control variables. You will find that if you know the relevant math, use of the higher-level packages is quite straightforward, easily learned from studying the corresponding DOC files.

We suggest that before commencing the Programming-mode lessons, you explore calculator-mode usage of the higher-level packages as far as your math background permits. Math curriculum sequences differ, but probably most users will be most comfortable trying the higher-level packaes in the approximate order EQN, SOLVE, ARRAY, MATRIX, LOG, TRGNEG, TRGPOS, DIF, INT and INTMORE. Since space becomes increasingly scarce as higher-level packages are loaded, you may have to reread file READ1ST.TXT to learn how to CONDENSE and SAVE if you haven't already.

Now for some parting advice about getting the most out of computer symbolic math:

First, storage and time consumption tends to grow dramatically with the number of variables in the input expressions, even if the ultimate result is fortuitously compact. For example, the number of terms in the expanded form of

$$(X1 + X2 + \ldots + XM) \hat{} N$$

grows outrageously with M and N. Consequently, it is important to make every effort to avoid needlessly introducing extra variables for generality's sake. Mathematical and physical problems are often stated using more variables than are strictly necessary, so it is also important to exploit every opportunity to reduce the number of variables from the original problem. Here are some general techniques for doing this:

1. If members of a set of variables can be made to occur only together as instances of a certain subexpression, consider replacing the subexpression with a single variable. For example:

   a) If K, X, and X0 can be made to occur only as instances of the subexpression K*(X-X0), then consider replacing this subexpression with a variable named perhaps KDX.

   b) Similarly, perhaps a combination such as M*C^2 could be replaced with E, or RHO*V^2/L could be replaced with RE.

   These are respectively instances of absorbing an offset together with a proportionality coefficient, renaming a physically-meaningful subexpression, and grouping quantities into dimensionless quantities. Most engineering and science libraries have books describing a more systematic technique called DIMENSIONAL ANALYSIS, and an article in the Journal of Computational Physics (June 1977) explains how computer algebra can automate the process.

2. Even when a variable cannot be eliminated, the complexity of expressions may be reduced if the variable can be made to occur only as instances of a subexpression. For example:

   a) If only even powers of a variable X occur, consider replacing X^2 with a variable named perhaps XSQ.

   b) If X only occurs as instances of 2^X, 2^(2*X), 2^(3*X),..., consider replacing 2^X with a variable named perhaps TWOTOX, yielding mere integer powers of that variable.

Some other advice is to avoid fractional powers and denominators as much as possible. They don't simplify well, they consume a lot of space, and they tend to be hard to decipher when printed one-dimensionally. Often a change in variable can eliminate a fractional power or a denominator.

Sometimes, even when a problem cannot be solved in its full generality, solving a few special cases enables one to infer a general solution which can perhaps then be verified by substitution or by induction. Alternatively, perhaps the original problem can be simplified by neglecting some lower-order contributions, in order to get an analytic solution which will at least convey some qualitative information about the solution to the original problem.

Sometimes only part of a problem or perhaps even none can be solved symbolically, and the rest must be solved numerically. If so, the attempt at an analytic solution at least allows one to proceed with an approximate numerical solution having more confidence that a concise analytical solution has not been overlooked. %

ECHO: #ECHO $ RDS () $

```
RFFIRST: 'RFFIRST $

MOVD (PRINT, #PRINT) $
FUNCTION PRINT (EX1),
  WHEN ATOM (EX1), #PRINT (EX1) EXIT,
  #PRINT (LPAR), PRINT (FIRST (EX1)), #PRINT (" . "),
  PRINT (REST (EX1)), #PRINT (RPAR), EX1,
ENDFUN $

MOVD (PRINTLINE, #PRINTLINE) $
FUNCTION PRINTLINE (EX1),
  PRINT (EX1), NEWLINE (), EX1,
ENDFUN $
ECHO: TRUE $
```

% This is the first of a sequence of interactive lessons about muSIMP programming. It presumes as a minimum that you have read section 8 of the muSIMP/muMATH Reference Manual. Also the first part of CLES1.ARI should be completed since it explains the mechanics of how to interact with the lessons. The file TRACE.MUS should be loaded.

muSIMP supplies a number of built-in functions and operators. The calculator-mode lessons introduced a few of these, such as RDS, RECLAIM, +, *, etc. These progamming-mode lessons introduce more built-in functions and operators, but more important, the lessons reveal how to supplement the built-in functions and operators with definitions of your own, thus extending muSIMP itself.

It is important to realize that, until the last programming-mode lessons, we will not deal with muMATH. Instead we deal first with muSIMP, the language in which muMATH is written. The illustrative examples for these first few lessons are utterly different from muMATH, because we want to suggest a few of the many other applications for which muSIMP is especially well suited, and because we want these lessons to be comprehensible regardless of math training level.

Data is what programs operate upon. The most primitive UNSTRUCTURED muSIMP data are integers and names, collectively called ATOMS to suggest their indivisibility by ordinary means. Some programs must distinguish these two types of atoms, so there are two corresponding RECOGNIZER functions: %

```
INTEGER (X76#) ;
NAME (X76#) ;
EG: -3271 $
INTEGER (EG) ;
NAME (EG) ;
```

% Do you suppose that "137", " ", "", and "X + 3", with the quotation marks included, are integers, names, or invalid? Find out for yourself%

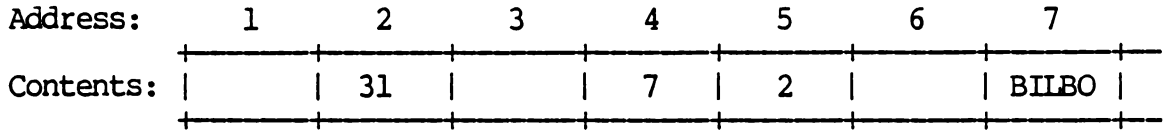
```
RDS: FALSE $
```

% Data can be stored in the computer's memory. The location at which a data item is stored is called its ADDRESS. An address is analogous to

a street address on the outside of a mailbox. The data stored there is analogous to mail inside the mailbox. As with a row of mailboxes, the contents of computer memory can change over time.

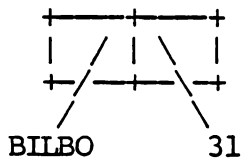
There are useful programs which deal only with unstructured data, but the most interesting applications involve aggregates of primitive data elements. One way to make an aggregate of 2 data elements is to use a structural data element called a NODE, which stores the addresses of the 2 data elements. Thus, a node is "data" consisting of addresses of 2 other data items.

For example, suppose that we wish to represent the aggregate consisting of the name BILBO and his age 31. We could store the name BILBO beginning at location 7, the number 31 beginning at location 2, and the node beginning at location 4. Then, begining at location 4, there would be stored the addresses 7 and 2, as illustrated in the following diagram:



Is that clear?

The specific placement of data within memory is managed automatically, so all we are concerned about is the specific name and number values and the connectivity of the aggregates. Thus, for our purposes it is best to suppress the irrelevant distracting detail associated with the specific addresses. The following diagram is one helpful way to portray only what we are concerned about:



This imagery suggests the word "pointers" for the addresses stored in nodes.

If you have seen one bisected box you have seen them all, so to reduce the clutter and thus emphasize the essential features, we henceforth represent such nodes by a mere vertex in our diagrams, giving schematics such as

Although most muSIMP programs use such aggregates internally, many muSIMP programs are designed to read and print data in whatever specialized notation is most suitable for the application. For example, muMATH uses operator and functional notation. Suppose however that we want to specify such aggregates directly in input and output. How can

we do it? If we have a nice graphics terminal, then then we conveniently could use diagrams such as the above. Most of us do not have nice graphics terminals, so we must use another external representation. For this purpose muSIMP uses a representation consisting of the first data item, followed by the second data item, separated by a dot and spaces, all enclosed in a pair of matching parentheses. For example:

```
(BILBO . 31)
```

We call this representation of a node a DOTTED PAIR. However, this is a diferent use of parentheses and periods from how they are otherwise used in muSIMP input, so we must preceed the dotted pair by the single-quote prefix operator to indicate to the parser that we are using dotted-pair notation rather than the usual operator or functional notation:

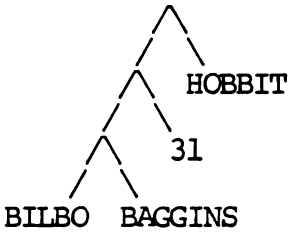
```
'(BILBO . 31)
```

Moreover, we must use an ampersand rather than a semicolon as the expression-terminator in order to inform the driver to print the expression as a dotted pair rather than attempt to print it using operator and functional notation. We say "attempt" because not all dotted pairs are appropriate for operator or functional printing, as we will explain in the last lessons. Here then is an example of dotted-pair input and printing: %

```
'(78 . TROMBONES) &
% Try a few of your own, and note what happens when you forget the
single-quote or use a semicolon rather than an ampersand: %
RDS: FALSE $
% What about when we want to represent an aggregate of more than two
atomic data elements? For example, what if we want to include BILBO's
last name, BAGGINS? Well, we can let one of the pointers of a node
point to another node, whose first pointer points to BILBO and whose
other pointer points to BAGGINS. For example:
```

We can input this as a dotted pair nested within a dotted pair: %

```
'((BILBO . BAGGINS) . 31) &
% Note that we only quote the outermost dotted pair. Now suppose that
we want to also include BILBO's species, structured as follows:
```



How would you input that?
Remember, your input must be terminated by an ampersand.
% RDS: FALSE $
% We would input it as: %

EG: '(((BILBO . BAGGINS) . 31) . HOBBIT) &

% An alternative structure for this information is the one corresponding to the input
'((BILBO . BAGGINS) . (31 . HOBBIT)).
On a piece of scratch paper, sketch the corresponding diagram, then hold it close to my face so I can check it.

% RDS: FALSE $
% My eyes must be getting bad, I couldn't see it. Oh well...

Since either element of a dotted pair can be a dotted pair, they can be used to represent arbitrary "binary tree structures". Moreover, although perhaps unprintable using pure dotted-pair notation, linked networks of binary nodes can be used to represent any data-structure whatsoever.

In order to do anything interesting with data aggregates, a program must be able to extract their parts. Accordingly, there are a pair of SELECTOR functions named FIRST and REST which respectively return the left and right pointers in a node. For example: %

REST (EG) &
FIRST (EG) &
FIRST (FIRST (EG)) &
REST (FIRST (EG)) &
% See if you can extract BILBO and BAGGINS from EG, using nested compositions of FIRST and/or REST: % RDS: FALSE $
% Our answers are: %
FIRST (FIRST (FIRST (EG))) &
REST (FIRST (FIRST (EG))) &
% Deeply nested function invocations become difficult to type and read, so let's define our first muSIMP function named FFFIRST, so that FFFIRST (EG) could be used as shorthand for the first of the above two examples and for any analogous example thereafter: %

FUNCTION FFFIRST (U),
  FIRST (FIRST (FIRST (U)))
ENDFUN &
% If you are not using a hard-copy terminal, jot down this function definition and all subsequent ones for reference later in the lesson.

Despite the word ENDFUN, the fun has just begun: Now that FFFIRST is defined, we can apply it at any subsequent time during the dialogue. For example: %
FFFIRST (EG) &

```
FFFIRST ('(((BIG . MAC) . CATSUP) . (FRENCH . FRIES))) &
```
% Using the definition of FFFIRST as a model, define a function named RFFIRST which extracts the REST of the FIRST of the FIRST of its argument, then test RFFIRST on EG: % RDS: FALSE $
% Our solution is: %

```
FUNCTION RFFIRST (FOO),
  REST (FIRST (FIRST (FOO))),
ENDFUN &
RFFIRST (EG) &
```
% The name FOO in the definition is called a PARAMETER, whereas EG where the function is applied is an example of an ARGUMENT. We can use any name for a parameter — even a name which has been bound to a value or even the same name as an argument. The name is merely used as a "dummy variable" to help indicate what to do to an argument when the function is subsequently applied. A function definition is like a recipe. It is filed away, without actually being EXECUTED until applied to actual arguments.

As another simple example, since an atom is defined as being either a name or an integer, it is convenient to have a recognizer function for atoms, so that we do not have to test separately for names and atoms when we do not care which type of atom is involved. We could define this recognizer as follows:

```
FUNCTION ATOM (U),
  NAME (U)  OR  NUMBER (U)
ENDFUN &
```

Actually, ATOM is already built-into muSIMP, but the example provides a good opportunity to introduce the built-in infix OR operator, which returns FALSE if both of its operands are FALSE, returning TRUE otherwise. Try out ATOM on the examples -5, X and EG % RDS: FALSE $
% Analogous to OR, there is a built-in infix AND operator which returns FALSE if either operand is FALSE, returning TRUE otherwise. There is also a built-in prefix NOT operator which returns TRUE if its operand is FALSE, returning FALSE otherwise. Knowing this, see if you can define a recognizer named NODE, which returns TRUE if its argument is a node, returning FALSE otherwise: % RDS: FALSE $
% In programming there is rarely, if ever, one unique solution, but ours is: %

```
FUNCTION NODE (U),
  NOT ATOM (U)
ENDFUN &
NODE (EG) &
NODE (5) &
```
% So much for trivial exercises. Now let's write a function which counts the number of atoms in its argument. We will count each instance of each atom, even if some atoms occur more than once.

At first this may seem like a formidable task, because a tree can be arbitrarily branched. How can we anticipate ahead of time all of these possibilities. Well, let's procrastinate by disposing of the most trivial cases even though we can't yet see the whole solution: If the

argument is an atom, then there is exactly 1 atom in it.

So much for trivial cases. We haven't yet solved the whole problem, but it builds our self-confidence to make progress, so that is a good psychological reason for first disposing of the easy cases. Also, with the easy cases out of the way, we can turn our full intellectual powers on the harder cases, unfettered by any distractions to trivial loose ends.

We are left with the case where we know we have a node. Perhaps we could somehow subdivide the problem into smaller cases?

Let's see ... Nodes have a FIRST part and a REST part, so perhaps that provides the natural subdivision. Hmmm ...

If we knew the number of atoms for the left part and the number for the right part, clearly the number for the whole aggregate is merely their sum. But how can we find out the number of atoms in these parts? Why not RECURSIVELY use the very function we are defining to determine these two contributions!

It may sound like cheating to refer to the function we are defining from with the definition itself, but remembering that the definition is not actually APPLIED until sometime after its definition is complete, perhaps it will work. We are working in a highly interactive environment, so the quickest way to resolve questions about muSIMP is to try it and see! Here then is a formal muSIMP function definition corresponding the the above informal English "algorithm": %

```
FUNCTION #ATOMS (U),
  WHEN ATOM (U), 1  EXIT,
  #ATOMS (FIRST(U))  +  #ATOMS (REST(U))
ENDFUN &
```

% Here we introduce 2 new concepts: The BODY of a function definition can consist of a sequence of one or more expressions separated by commas. A CONDITIONAL-EXIT is an expression consisting of a sequence of one or more expressions nested between the matching pair of words WHEN and EXIT. When a function definition is APPLIED, the expressions in its body are evaluated sequentially, until perhaps a conditional exit causes an exit from the procedure or until the delimiter named ENDFUN is reached. For a conditional exit, the first expression after the word WHEN is evaluated. If the value is FALSE, then evaluation proceeds to the point immediately following the matching delimiter named EXIT. Otherwise, evaluation proceeds sequentially through the remaining expressions in the conditional exit, if any, exactly as if the body of the conditional exit replaced that of the function. The value of a conditional exit is that of the last expression evaluated therein, and the value returned by a function is that of the last expression evaluated therein when the function is applied.

Thus, #ATOMS immediately returns the value 1 whenever the argument is an atom, and otherwise the function breaks the problems into two parts which are necessarily smaller, hence closer to being atoms. Let's test it, starting with trivial cases first: %

```
#ATOMS (FOO) &
#ATOMS (5) &
EG &
#ATOMS (EG) &
```
% It looks promising, but it is still perhaps mysterious how muSIMP and #ATOMS keep track of all of these recursive function invocations. Since the trace package is supposedly loaded, to trace the execution of #ATOMS, we merely issue the command: %

```
TRACE (#ATOMS) &
```
% Now every time #ATOMS is entered, it prints its name and argument values, whereas every time it is exited, it prints its name followed by an equal sign, followed by the returned value. Moreover, the trace is indented in a manner which allows corresponding entries and exits to be visually associated. Watch: %

```
#ATOMS (FOO) &
EG &
#ATOMS (EG) &
```
% Try a few examples of your own, until these new ideas begin to gel:
% RDS: FALSE $

```
UNTRACE (#ATOMS) &
#ATOMS (FOO) &
```
% Here is a function which counts only the number of integers in its argument: %

```
FUNCTION #INTEGERS (U),
  WHEN INTEGER (U), 1 EXIT,
  WHEN NAME (U), 0 EXIT,
  #INTEGERS (FIRST(U)) + #INTEGERS (REST(U))
ENDFUN $
EG &
#INTEGERS (EG) ;
```
% Now, using it as a model, try writing a function named #NAMES, which returns the number of names in its argument. If your first syntactically accepted attempt fails any test, try using TRACE to reveal the reason why: % RDS: FALSE $
% Our solution is ...

On second thought, we won't give you our solution. Consequently, if you were lazy and didn't try, you had better try now, because the examples will get steadily harder now. % RDS: FALSE $
% The HEIGHT of an atom is 1, and the HEIGHT of a node is 1 more than the maximum of the two heights of its FIRST and REST parts. Accordingly, let's first write a function named MAX, which returns the maximum of its two integer arguments. There is a built-in infix integer comparator named ">", so here is a hint:

```
FUNCTION MAX (INT1, INT2),
  WHEN INT1 > INT2, ... EXIT,
  ...
ENDFUN $
```

Enter such a definition, with appropriate substitutions for the missing

```
portions, then test your function to make sure it works correctly:  %
RDS: FALSE $
%   Now, with the help of our friend MAX, see if you can write a
function named HEIGHT, which returns the height of its argument: %
RDS: FALSE $
%  Our solution is: %

FUNCTION HEIGHT (U),
  WHEN ATOM (U),  1  EXIT,
  1 + MAX (HEIGHT(FIRST(U)), HEIGHT(REST(U)))
ENDFUN $
%   This brings us to the end of the first programming-mode lessons.  It
may be a good idea to review this lesson before proceeding to lesson
PLES2.TRA.  %

ECHO: FALSE $
MOVD (#PRINT, PRINT) $  MOVD (#PRINTLINE, PRINTLINE) $
RDS () $
```

```
MOVD (PRINT, #PRINT) $
FUNCTION PRINT (EX1),
  WHEN ATOM (EX1), #PRINT (EX1) EXIT,
  #PRINT (LPAR), PRINT (FIRST(EX1)), #PRINT (" . "),
  PRINT (REST(EX1)), #PRINT (RPAR), EX1,
ENDFUN $

MOVD (PRINTLINE, #PRINTLINE) $
FUNCTION PRINTLINE (EX1),
  PRINT (EX1), NEWLINE (), EX1,
ENDFUN $

ECHO: TRUE $
```

% This is the second of a sequence of muSIMP programming lessons.

EQ is a primitive muSIMP Comparator operator which returns TRUE if its two operands are the same object or equal integers, returning FALSE otherwise: %

```
FIVE: 5 $  5 EQ FIVE ;
```

% Names are stored uniquely, so two occurences of a name must involve the same address: %

```
ACTOR: 'BOGART ;  ACTOR EQ 'BOGART ;
```

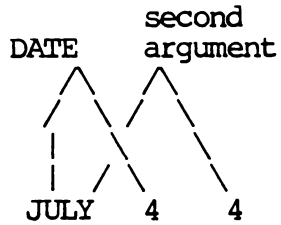
% Here is an example of two different references to the same physical node: %

```
DATE: '(JULY . 4) &  FOO:  DATE $  FOO EQ DATE ;
```

% However, watch this: %

```
DATE EQ '(JULY . 4) ;
```

% What happened? The two aggregates are DUPLICATES, but since they were independently formed they do not start with the same node. In fact, only the name JULY is shared among them, as shown below:



Clearly it is desirable to have a more comprehensive equality comparator which also returns TRUE for aggregates which are duplicates in the sense of printing similarly. Let's write such a function, called DUP. Following the general advice given in PLES1, let's first dispose of the trivial cases:

If either argument is an atom, then they are duplicates if and only if they are EQ.

Otherwise, they are both nodes, which is the nontrivial case. Now, let's employ our "divide-and-conquer" strategem, using FIRST and REST as the partitioning. Two nodes refer to duplicate aggregates if and only if the FIRST parts are duplicates and the REST parts are duplicates. Moreover, that can be tested with our beloved recursion, using DUP itself!

See if you can write a corresponding function named DUP: %
RDS: FALSE $
% There are many possible variants, but here is one of the most compact: %

```
FUNCTION DUP (U, V),
  WHEN ATOM (U),  U EQ V  EXIT,
  WHEN ATOM (V),  FALSE  EXIT,
  WHEN DUP (FIRST(U), FIRST(V)),  DUP (REST(U), REST(V)) EXIT,
ENDFUN $
```

% An interesting challenge for your spare time is to see how many different but reasonable ways this function can be written.

Actually, there already is a built-in infix operator named "=", which is equivalent to DUP: %

```
DATE: '(JULY . 4) $
DATE = '(JULY . 4) ;
```

% Do you feel DUPed to learn that an exercise duplicated an existing facility?

It is crucial to understand exactly what the existing facilities do, and the best way to learn that is to understand how they work by creating them independently.

Here is a good exercise: See if you can write a comparator function named SAMESHAPE, which returns TRUE if its two arguments are similar in the sense of having nodes and atoms at similar places. For example,

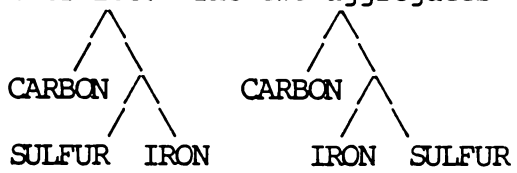
```
SAMESHAPE ('((KINGS . ROOK) . 5), '((QUEENS . 3) . PAWN))
```

is TRUE: % RDS: FALSE $
% This is one of those instances where we will not give the answer.

Now, using the infix operator named "=", see if you can write a function named CONTAINS which returns TRUE if its first argument is a duplicate of its second argument or contains a duplicate of its second argument. For example,

((JULY . 4) . (1931 . FRIDAY))

contains (1931 . FRIDAY). It is at least as hard as DUP, so take your time and don't give up easily. % RDS: FALSE $
% Here is a harder exercise: The two aggregates



are ISOMERS because they are either the same atom or at every level either the left branches are isomers and the right branches are isomers,

or the left branch of one is an isomer of the right branch of the other and vice-versa. Write a corresponding comparator function named ISOMERS. (It's similar to DUP, with a twist.) % RDS: FALSE $
% Our answer is: %

```
FUNCTION ISOMERS (U, V),
  WHEN ATOM (U), U EQ V EXIT,
  WHEN ATOM (V), FALSE EXIT,
  ISOMERS (FIRST(U), FIRST(V)) AND ISOMERS (REST(U), REST(V))
    OR ISOMERS (FIRST(U), REST(V)) AND ISOMERS (REST(U), FIRST(V))
ENDFUN $
```

% Because of all the combinations which might have to be checked, the execution time for this function can grow quite quickly with depth. Try tracing a few examples of moderate depth: % RDS: FALSE $
% So far our functions have merely dismantled or analyzed aggregates given to them as arguments. None of our examples have constructed new aggregates. The dot of course results in aggregates, but this occurs as the dot is read. Moreover, since the single quote necessarily preceeding an outermost dotted pair prevents evaluation, bound variables in a dotted pair contribute merely their names rather than their values. For example: %

```
EG: 7 $  '(EG . 3) &
```

% What we want is a function which evaluates its two arguments in the usual way, then returns a node whose two pointers point to those values. There is such a function, named ADJOIN: %

```
ADJOIN (EG, 3) &
```

% A dotted pair within a function definition is a static entity, frozen at the time the function is defined. In contrast, a reference to ADJOIN within a function definition is dynamic. The node creation is done afresh, with the current values of its arguments every time that part of the function is applied. As an example of the use of ADJOIN, let's write a function named SKELETON, which constructs a new tree which is structurally similar to its argument but has the name of length zero, "", wherever its argument has an atom. Thus, when printed, the new aggregate will display the skeletal structure of the aggregate without visually-discernable atoms. For example,
SKELETON ('((HALLOWEEN . GHOSTS) . WITCHES)) & will yield ((.) .)

OK, let's recite the litany: What comes first?

TRIVIAL CASES.

So, if the argument is an atom we return what?

"".

Otherwise we have a node, which is the most general case. However, nodes have a FIRST and a REST, so can we somehow recurse, using SKELETON on these parts, then combine them?

Yes, as follows: %

```
FUNCTION SKELETON (U),
  WHEN ATOM (U),  ""  EXIT,
  ADJOIN (SKELETON (FIRST(U)), SKELETON (REST(U)))
ENDFUN $
SKELETON ('((MOO . GOO) . (GUY . PAN))) &
% Easy.  Yes?
```

Now it is your turn. Write a function named TREEREV, which produces a copy of its argument in which every left and right branch are interchanged at every level. For example,

```
          TREEREV ('((MOO . GOO) . (GUY . (PAN . CAKE)))) &
```

should yield

```
                  (((CAKE . PAN) . GUY) . (GOO . MOO))
% RDS: FALSE $
```

% If you didn't get the following solution, you may groan when you see how easy it is: %

```
FUNCTION TREEREV (U),
  WHEN ATOM (U),  U  EXIT,
  ADJOIN (TREEREV (REST(U)),  TREEREV (FIRST(U)))
ENDFUN &
TREEREV ('(("Isn't" . that) . easy)) &
```

% Here is a somewhat harder exercise: Write a function named SUBST, which returns a copy of its first argument wherein every instance of its second argument is replaced by its third argument. For example, if

```
PHRASE:
  '(((THIS . (GOSH . DARN)) . CAR) . (IS . ((GOSH . DARN) . BAD))) $
```

then SUBST (PHRASE, '(GOSH . DARN), '(expletive . deleted)) yields

```
(((THIS . (expletive . deleted)) . CAR)
   . (IS . ((expletive . deleted) . BAD)))  %          RDS: FALSE $
% That's all folks.
```

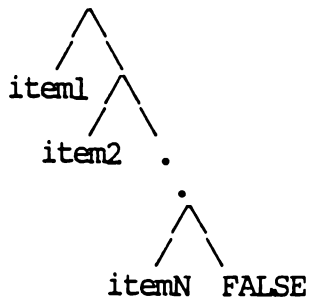
The next lesson deals with a special form of tree called a list. Many people find lists more to their liking, and perhaps you will too.%

```
ECHO: FALSE $
MOVD (#PRINT, PRINT) $  MOVD (#PRINTLINE, PRINTLINE) $
RDS () $
```

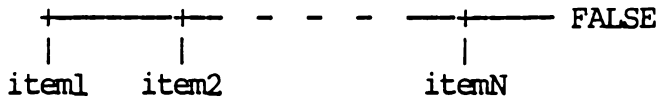
ECHO: TRUE $

% This is the third in a series of muSIMP programming lessons.

Often, it is most natural to represent a data aggregate as a sequence or LIST of items rather than as a general binary tree. For example, such a sequence is quite natural for the elements of a vector or of a set. We can represent such a sequence in terms of nodes by having all of the FIRST cells point to the data elements, using the REST cells to link the sequence together. The last linkage node can have a REST cell which is FALSE to indicate that there are no further linkage nodes:



When this diagram is rotated 45 degrees in the counter-clockwise direction, it looks like a clothes line with the successive data elements suspended from it.



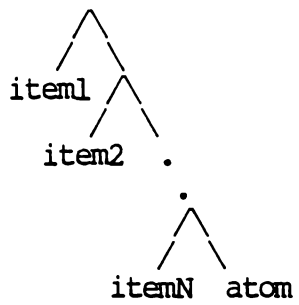
This latter diagram more clearly suggests a sequence or list of items. The simple regularity of the structure permits correspondingly simple function definitions for processing such structures. Moreover, the linear structure suggests an external printed representation which is far more readable than dotted pairs. In response to an ampersand terminator, muSIMP prints the above aggregate in the more natural LIST notation:

(item1 item2 ... itemN)

rather than the equivalent dot notation

(item1 . (item2 . ... (itemN . FALSE) ...))

Conversely, the reader accepts list notation as an alternative input form to dot notation. Naturally, any of the items in a list can themselves be either lists or more general dotted pairs. The printer uses list notation as much as possible. Thus, a structure of the form



where "atom" is not the atom FALSE, is printed in a mixed notation as

(item1 item2 ... (itemN . atom))

The muSIMP read routines will correctly read such mixed notation. You may wonder why you never noticed list notation being output in PLES1 and PLES2. At the beginning of those lessons the function PRINT was redefined so that it printed expression entirely in dot notation.

It is important to fully understand the connection between dotted pairs and lists, so take 5 minutes or so to type in some lists, nested lists, nested dotted pairs, and mixtures, noting carefully how they print. % RDS: FALSE $
% Did your examples include: %

'() &
% Is that surprising? Since FALSE is used to signal the end of the list, FALSE and the empty list must be equivalent. %

FALSE EQ '() &
% Clearly functions which successively process each element of a list must somehow determine when the end of the list has been reached. This TERMINAL CASE is easily achieved by an equality test for the name FALSE. Since the need for this test is so pervasive in muSIMP, the empty list recognizer EMPTY is written in machine language for efficiency reasons. However, it could be defined using an EQ test as follows:

```
FUNCTION EMPTY (LIS),
  LIS EQ '(),
ENDFUN;
```

Using EMPTY, see if you can define a function named #ITEMS, which returns the number of (top-level) items in its list argument. For example, #ITEMS ('(FROG, (FRUIT . BAT), NEWT)) should yield 3. Here is an incomplete solution. All you have to do is enter it with the portions marked "..." appropriately filled.

```
FUNCTION #ITEMS (LIS),
  WHEN EMPTY (LIS), ...  EXIT,
  1 + #ITEMS ( ... )
ENDFUN $
% RDS: FALSE $
```

% Actually, there is already a built-in function called LENGTH, which returns the length of a list. It is somewhat more general in that it returns the number of characters necessary for printing when given an atom.

Note that with lists it is typical to recur only on the REST of the list, whereas with general binary trees it is typical to recur on both the FIRST and the REST.

So far, the examples and exercises have been relatively isolated ones. Now we will focus on writing a collection of functions which together provide a significant applications package:

A list provides a natural representation for a set. For example, (MANGO, (CHOCOLATE . FUDGE), (ALFALFA, SPROUTS)) can represent a set of three foods. Using this representation, let's write functions which test set membership and form unions, intersections, etc.

First, write a function named ISIN, which returns TRUE if its first argument is in the list which is its second argument, returning FALSE otherwise: % RDS: FALSE $
% Our solution is: %

```
FUNCTION ISIN (U, LIS),
  WHEN EMPTY (LIS), FALSE EXIT,
  WHEN U = FIRST (LIS), EXIT,
  ISIN (U, REST(LIS))
ENDFUN $
ISIN ('FROG, '(SALAMANDER NEWT TOAD)) ;
```

% Actually, there is already a built-in version of ISIN called MEMBER.

A set contains no duplicates, so we really should have a recognizer function named ISSET, which returns TRUE if its list argument contains no duplicates, returning FALSE otherwise. Try to write such a function:
% RDS: FALSE $
% Here is a hint, in case you gave up:

```
FUNCTION SET (LIS),
  WHEN ... EXIT,
  WHEN MEMBER (FIRST(LIS), ... ), FALSE EXIT,
  SET ( ... )
ENDFUN;                          %       RDS: FALSE $
```

% In case it isn't clear by now, a rule of this game is that you are free (and encouraged) to use any functions we have already discussed, whether they are built-in, previous examples, or previous exercises. That is one reason it is adviseable for you to actually do the exercises.

Now write a function named SUBSET, which returns TRUE if the set which is its first argument is a subset of that which is its second argument. (Remember that every set is a subset of itself and the empty set is a subset of every set.) % RDS: FALSE $
% Here is a hint, in case you gave up or had a less compact solution:

```
FUNCTION SUBSET (SET1, SET2),
  WHEN ... EXIT,
  WHEN MEMBER (FIRST(SET1), ...),  SUBSET( ...)  EXIT
ENDFUN;                          %       RDS: FALSE $
```

% Two sets are equal if and only if they contain the same elements. However, the elements need not occur in the same order. Write a corresponding comparator function named EQSET: % RDS: FALSE $

% Ah yes, a hint perhaps?:

```
FUNCTION EQSET (SET1, SET2),
   ...
ENDFUN;          %       RDS: FALSE $
```

% Do you think that's not much of a hint?

Well, the body of the function really can be written with one modest line, so try harder: % RDS: FALSE $

% Remember the rules of the game: You are encouraged to use any function discussed previously: %

```
FUNCTION EQSET (SET1, SET2),
  SUBSET (SET1, SET2)  AND  SUBSET (SET2, SET1)
ENDFUN;
```

% Our examples so far have merely analyzed sets. We can use ADJOIN to construct lists, just as we used ADJOIN to construct binary trees. As an example of this, write a function named MAKESET, which returns a copy of its list argument, except without duplicates if there are any: % RDS: FALSE $

% If you need a hint, here is one, but it is all you will get:

```
FUNCTION MAKESET (LIS)
  WHEN ...,  '()  EXIT,
  WHEN MEMBER ( ... ), ...  EXIT,
  ADJOIN ( ... )
ENDFUN;          %       RDS: FALSE $
```

% Let's see if your solution works correctly: %

```
MAKESET ('(FROG, FROG, FROG)) &
```

% If there is a duplicate in the answer, then back to the computer terminal: % RDS: FALSE $

% (It helps to think of nasty test cases BEFORE you start programming).

Now for the crowning glory of our set package: The UNION of two sets is defined as the set of all elements which are in either (perhaps both) sets. Give it a try: % RDS: FALSE $

% A hint perhaps? Well, the function body can be written in 3 lines, each of which begins just like the corresponding line in our hint for MAKESET. % RDS: FALSE $

% Here is our solution: %

```
FUNCTION UNION (SET1, SET2),
  WHEN EMPTY (SET1),  SET2  EXIT,
  WHEN MEMBER (FIRST(SET1), SET2),  UNION (REST(SET1), SET2)  EXIT,
  ADJOIN (FIRST(SET1), UNION (REST(SET1), SET2))
ENDFUN $
```

```
UNION ('(DOG, CAT, 5, RAT), '(-5, CAT, PIG, DOG))&
```
% The intersection of two sets is the set of all elements which are in both sets. Using our definition of UNION as inspiration, write a corresponding function for the intersection: % RDS: FALSE $
% So far, our set algebra package has been developed in a so-called BOTTOM-UP manner, with the most primitive functions defined first, and with the more sophisticated functions defined in terms of them. The opposite approach is TOP-DOWN, where we define the most comprehensive functions in terms of more primitive ones, then we define those more primitive ones in terms of still more primitive ones, until no undefined functions remain.

As an example of the top-down attitude, let's write a SYMMETRIC DIFFERENCE function for our set-algebra package. The symmetric difference of two sets is the set of all elements which are in exactly one of the two sets. This is in contrast to the ordinary diference of two sets, which is all of the elements that are in the first set but not the second. However, if an ordinary difference function was available, we could write the symmetric difference as the union of the ordinary difference between set1 and set2, with the ordinary difference between set2 and set1. We have already written UNION, but an ordinary set difference is not yet available. Nevertheless, let's bravely proceed to write the symmetric difference in terms of the ordinary difference, then we will worry about how to write the latter:
%

```
FUNCTION SYMDIF (SET1, SET2),
  UNION (ORDDIF (SET1, SET2),  ORDDIF (SET2, SET1))
ENDFUN $
```
% Now you try to write ORDDIF. It may help you to know that it can be written very similarly to UNION: % RDS: FALSE $
% Some programmers are initially uncomfortable with the top-down approach because it makes them nervous to refer to undefined functions: there are obvious loose ends during the writing process. However, it is not necessary to understand how an auxiliary function can be written before daring to refer to it. All that is necessary is that the duty relegated to the auxiliary function be somehow more elementary than the overall duty performed by the function which refers to it.

There are necessarily loose ends during the writing of a program in any sequential order. With the bottom-up approach, the loose ends are neither written nor referred to until lower-level functions have been written. Unfortunately, as such hidden loose ends are revealed they often make apparent the need to completely reorganize and rewrite all subordinate functions into a more suitable organization. In contrast, the obvious loose ends during a top-down development provide invaluable clues about how to organize the remaining functions. Moreover, any subsequent changes tend to be easier, because communication between the functions is more localized, more independent, and more hierarchial. For example, we know that in the definition of SYMDIF we are taking the union of two DISJOINT sets, because from the definition of ORDDIF it is clear that ORDDIF (SET1, SET2) and ORDDIF (SET2, SET1) cannot have elements in common. Hence it would be more efficient merely to append the second ordinary set difference to the first ordinary set difference,

or vice-versa. Unfortunately, ADJOIN does not accomplish the desired effect.

For example, ADJOIN ('(5, 9), '(3, 7)) yields ((5, 9), 3, 7) rather than the desired (5, 9, 3, 7). What we must do is ADJOIN 9 to (3, 7), then adjoin 5 to that result. See if you can generalize this process into a function named APPEND, which returns a list consisting of the list which is its first argument appended onto the beginning of the list which is its second argument:% RDS: FALSE $
% How about: %

```
FUNCTION APPEND (LIS1, LIS2),
  WHEN EMPTY (LIS1), LIS2 EXIT,
  ADJOIN (FIRST(LIS1), APPEND (REST(LIS1), LIS2))
ENDFUN $
```

% You may not be getting tired, but my circuits are weary, so let's bring this lesson to a close. %

```
ECHO: FALSE $ RDS () $
```

```
ECHO: TRUE $
```
% This is the fourth in a series of muSIMP programming lessons.

Often within a function definition it is necessary to DYNAMICALLY create a list. Suppose we want to make a list of the values of the variables FIRSTNAME, LASTNAME, and MAILADDRESS. It will not do to use the program statement

```
'(FIRSTNAME, LASTNAME, MAILADDRESS),
```

because the quote operator prevents evaluation of the variables. However, the desired effect can be achieved by the statement

```
ADJOIN (FIRSTNAME, ADJOIN (LASTNAME, ADJOIN (MAILADDRESS, '()))).
```

muSIMP provides the function LIST to achieve this effect much more compactly and conveniently. Thus, the above list could be created with the following statement:

```
LIST (FIRSTNAME, LASTNAME, MAILADDRESS).
```

Unlike most functions, LIST can have any arbitrary number of arguments. For example consider the following assignments: %

```
FIRSTNAME: 'JOHN &
LASTNAME: 'DOE &
MAILADDRESS: 'TIMBUKTU &
```
% Create a list of these variables using the quote operartor and compare it with a list created using the function LIST: % RDS: FALSE $
% A useful utility to have is a constructor function which reverses a list. Writing such a function can be somewhat tricky. The following skeletal definition uses our friends APPEND and LIST as helper functions:

```
FUNCTION REVLIS (LIS),
  WHEN  ... , FALSE  EXIT,
  APPEND ( ... ,  LIST (FIRST (LIS))),
ENDFUN $
```

See if you can successfully complete this definition. Naturally, you also have to reenter APPEND if a correct version is not around from the previous lesson. (Remember to jot down all function definitions if you are not using a hard-copy terminal.) % RDS:FALSE $
% A well-written APPEND necessarily requires execution time which is approximately proportional to the length of its first argument. The REVLIS function outlined above invokes APPEND n times if n is the length of its original argument, and the average length of the argument to APPEND is n/2. Thus, the time is approximately proportional to n*(n/2), which is proportional to n^2.

An technique using COLLECTION VARIABLES permits a list to be reversed in time proportional to n, yielding tremendous time savings for

long lists: %

```
FUNCTION REVLIS (LIS, ANS),
  WHEN EMPTY (LIS),  ANS  EXIT,
  REVLIS (REST(LIS),  ADJOIN (FIRST(LIS), ANS))
ENDFUN $
TRACE (REVLIS) &
REVLIS ('(1, 2, 3)) &
```

% A collection variable accumulates the answer during successive recursive invocations. Then, the resulting value is passed back through successive levels as the returned answer.

As is illustrated here, we can invoke a function with fewer arguments than there are parameters. When this is done, the extra parameters are initialized to FALSE, and they are available for use as LOCAL VARIABLES within the function body. Quite often, as in this example, the initial value of FALSE is exactly what we want, because it also represents the empty list. (When we want some other initial value, either the user can supply it, or the function can supply it to an auxiliary function which does the recursion.)

Of course, if a user of REVLIS supplies a second argument, then the function returns the reversed first argument appended onto the second argument. This "feature" is occasionally quite useful.

What if the user supplies more arguments than there are parameters? The extra arguments are evaluated, but ignored.

Up to this point the lessons have taught the "applicative" style of programming. The emphasis has centered on expression evaluation, functional composition, and recursion. The power and elegance of applicative programming was the topic of an influential Turing Lecture by J Backus. The lecture was published in the August 1978 issue of the Communications of the ACM.

muSIMP also supports the alternative "Von Neumann" style emphasizing loops, assignments, and other side-effects. To illustrate this style, here is an alternative definition of REVLIS which introduces the LOOP construct: %

```
FUNCTION REVLIS (LIS, ANS),
  LOOP
    WHEN EMPTY (LIS),  ANS  EXIT,
    ANS: ADJOIN (FIRST(LIS), ANS),
    LIS: REST (LIS)
  ENDLOOP
ENDFUN $
```

% muSIMP has a function named REVERSE which is defined in machine language. Since it is entirely equivalent to REVLIS and much faster, REVERSE should normally be used in place of REVLIS in application programs written by the user.

An iterative loop is an expression consisting of the keyword LOOP, followed by a sequence of one or more expressions separated by commas, followed by the matching delimiter named ENDLOOP. The body of a loop is

evaluated similarly to a function body, except:

1. When evaluation reaches the delimiter named ENDLOOP, evaluation proceeds back to the first expression in the loop.

2. When evaluation reaches an EXIT within the loop, evaluation proceeds to the point immediately following ENDLOOP, and the value of the loop is that of the last expression evaluated therein.

There can be any number of conditional exits anywhere in a loop. Ordinarily there is at least one exit unless the user plans to have the loop repeat indefinitely. Now consider the following sequence: %

```
L1: '(THE ORIGINAL ) $
L2: '(TAIL) $
LIS: 'DOG &
ANS: 'CAT &
REVLIS (L1, L2) &
```

% The above definition of REVLIS makes assignments to its parameters LIS and ANS. For this example, the final assignments are LIS: '() and ANS: '(ORIGINAL, THE, TAIL). So, what do you guess are the corresponding current values for LIS and ANS? See for yourself: %

```
RDS: FALSE $
```

% The assignments to parameters LIS and ANS in REVLIS has no effect on their values once the function has returned! The restoration of the original environment following the return from a called function allows the programmer to change the value of a function's parameters without fear of damaging the values the parameters of the same name have outside the function. Thus functions can be thought of as "black boxex" which have no effect other than their returned value.

A function's parameters can not be used to pass information back to the calling function. If we wish to return more than one piece of information, a list of values can be returned. However, another way is to make assignments within the function body to variables which are not among its parameters. Such variables are called "fluid" or "global" variables.

The iterative version of REVLIS using the LOOP construct is slightly faster than the recursive version, but the latter is more compact. When there is such a trade-off between speed and compactness, a good strategy is to program for speed in the crucial most frequently used functions, and program for compactness elsewhere.

Another consideration when choosing between iteration and recursion is the amount of storage required to perform a given task. Each time a function is called information must be stored on a STACK so the original environment can be restored when the function returns. Since recursion involves the nesting of function calls, a highly recursive function can exhaust all available memory before completing its task. This will result in the

ALL Spaces Exhausted

error trap. The use of iteration in this situation might permit an equivalent computation to proceed to termination.

For practice with loops, use one to write a nonrecursive recognizer named ISSET, which returns TRUE if its list argument contains no duplicate elements, returning FALSE otherwise. (Compare your definition with the recursive version in lesson PLES3.) % RDS: FALSE $
% Here is our solution: %

```
FUNCTION ISSET (LIS),
  LOOP
    WHEN EMPTY (LIS), EXIT,
    WHEN MEMBER (FIRST(LIS), REST(LIS)),  FALSE  EXIT,
    LIS: REST (LIS)
  ENDLOOP
ENDFUN $
ISSET ('(DOG, CAT, COW, CAT, RAT)) &
```

% Another good exercise adapted from PLES3 is to use a loop to write a nonrecursive function named SUBSET, which returns TRUE if its first argument is a subset of its second argument, returning FALSE otherwise:
% RDS: FALSE $
% A BLOCK is another control construct which is sometimes convenient, particularly in conjuction with the Von Neumann style. As an illustration of its use, the following iterative version of the MAKESET function from PLES3 returns a set composed of the unique elements in the list which is its first argument: %

```
FUNCTION MAKESET (LIS, ANS),
  LOOP
    WHEN EMPTY (LIS),  ANS  EXIT,
    BLOCK
      WHEN MEMBER (FIRST(LIS), ANS),  EXIT,
      ANS:  ADJOIN (FIRST(LIS), ANS)
    ENDBLOCK,
    LIS: REST (LIS)
  ENDLOOP
ENDFUN $
MAKESET ('(FROG, FROG, FROG, TERMITE)) &
```

% When evaluation reaches an EXIT, it proceeds to the point following the next ENDBLOCK, ENDLOOP, or ENDFUN delimiter — whichever is nearest. Thus, BLOCK provides a means for alternative evaluation paths which rejoin within the same function body or loop body, without causing an exit from that body. The first expression in a block must be a conditional-exit (anything else can be moved outside anyway), but since there can be any number of other conditional exits or other expressions within the block, the block provides a very general structured control mechanism. For example, the CASE-statement and IF-THEN-ELSE construct of some other languages are essentially special cases of a block.

You may not have noticed, but the loop version of MAKESET has the effect of reversing the order of the set elements. Using ADJOIN in a loop generally has this effect, which is why it is so suitable for REVERSE. With sets, incidental list reversal is perhaps acceptable, but for most applications of lists it is not. We could of course use a preliminary or final invocation of REVERSE so that the final list would emerge in the original order, but that would relinquish the speed advantage of the loop approach, while further increasing its greater

bulk. Thus, recursion is usually preferable to loops when ADJOIN is involved. For example, recursion is used almost exclusively to implement muMATH, because its symbolic expressions are represented as ordered lists.

Loops are also less applicable to general tree structures than to lists, but it is often possible to loop on the REST pointer while recursing on the first pointer, or vice-versa, particularly if ADJOIN is not involved. For example, compare the following semi-recursive definition of #ATOMS with the fully-recursive one in PLES1: %

```
FUNCTION #ATOMS (U, N),
  N: 1,
  LOOP
    WHEN ATOM (U),  N  EXIT,
    N: N + #ATOMS (FIRST(U)),
    U: REST (U)
  ENDLOOP
ENDFUN $
#ATOMS ('((3 . FOO),  BAZ)) &
```

% If the answer surprises you, don't forget the FALSE which BAZ is implicitly dotted with.

See if you can similarly write a semi-recursive function named DUP which does what the infix operator named "=" does: % RDS: FALSE $

% Those of you with previous exposure to only Von Neumann style programming undoubtedly feel more at home now. The reason we postponed revealing these features until now is that we wanted to force the use of applicative programming long enough for you to appreciate it too. Naturally, one should employ whichever style is best suited for each application, so it is worthwhile to become equally conversant with both styles.

Thus endeth the sermon. %

```
ECHO: FALSE $  RDS () $
```

ECHO: TRUE $

% This is the fifth in a series of muSIMP programming lessons.

In the previous lesson our original version of REVERSE, called REVLIS, required time proportional to n^2, where n is the length of the first argument. We then showed how a collection variable or a loop could yield a much faster technique using time proportional only to n. Now, let's consider the speed of some of the other set functions that we defined:

Whether iterative or recursive, MEMBER can require a number of equality comparisons equal to the length of its second argument. Whether defined iteratively or recursively, SUBSET, EQSET, UNION, and INTERSECTION all require a membership test for each element of one argument in the list which is the other argument. Thus, these definitions can all consume computation time which grows as the product of the lengths of the two arguments. By similar reasoning, the one-argument functions ISSET and MAKESET are seen to require time proportional to the square of the length of their argument. Data-base applications and others can involve thousands of set operations on sets having thousands of elements, so it is worthwhile to seek methods for which the computation time grows more slowly with set size.

In muSIMP, every name has an associated PROPERTY LIST which is immediately accessible in an amount of time that is independent of the total number of names in use. Provided the elements of the sets are all names, this permits techniques for the above set operations requiring time proportional merely to the length of the one set or to the sum of the lengths of the two sets.

A property list is a list of dotted pairs. The first of each dotted pair is an expression called the KEY or INDICATOR, and the rest of each dotted pair is an expression called the associated INFORMATION. For example, in a meteorological data-base application, the name HONOLULU might have the property list

((RAIN . 2), (HUMIDITY . 40), (TEMPERATURE, 58, 96))

The function used in the form GET (name, key) returns the information which is dotted with the value of "key" on the property list of the value of "name", returning FALSE if no such key occurred on the property list.

A command of the form PUT (name, key, information) causes the value of "key" dotted with the value of "information" to be put on the property list of the value of "name". PUT returns the value of "information".

Using property lists, the basic technique for accomplishing our various operations on two sets of names is:

1. For each name in one of the two sets of names, store TRUE under the key SEEN.

2. For each name in the other set, check to determine whether or not the name has already been seen, and act accordingly.

3. For each name in the first set, remove the property SEEN so that we won't invalidate subsequent set operations which utilize any of the same elements.

A simpler variant of this idea is applicable to the one-argument functions named ISSET and MAKESET.

As an example, here is UNION defined using this technique together with the applicative style: %

```
FUNCTION UNION (SET1, SET2),
  MARK (SET1),
  UNMARK (SET1, UNIONAUX (SET2))   ENDFUN $
FUNCTION MARK (SET1),
  WHEN EMPTY (SET1),  EXIT,
  PUT (FIRST(SET1), 'SEEN, TRUE),
  MARK (REST (SET1))   ENDFUN $
FUNCTION UNIONAUX (SET2),
  WHEN EMPTY (SET2),  SET1  EXIT,
  WHEN GET (FIRST(SET2), 'SEEN),  UNIONAUX (REST(SET2))  EXIT,
  ADJOIN (FIRST(SET2), UNIONAUX(REST(SET2)))   ENDFUN $
FUNCTION UNMARK (SET1, ANS),
  WHEN EMPTY (SET1),  ANS  EXIT,
  PUT (FIRST(SET1), 'SEEN, FALSE),
  UNMARK (REST(SET1), ANS)   ENDFUN $
UNION ('(A, B, C, D), '(F, A, E, C)) &
```

% Each time any function is invoked, the outside values of its parameter names, if any, are "stacked" away to be restored later, just prior to return from that invocation. If a function refers to a variable which is not among its parameters, then the most recent value of the variable on the stack is used. Thus, when UNIONAUX is invoked from within UNION, SET1 in the definition of UNIONAUX refers to the argument value associated with that parameter of UNION. This treatment is called "dynamic binding", and a reference such as to SET1 in UNIONAUX is called a "fluid reference". We could have avoided this by making SET1 be an argument and a parameter to UNIONAUX, but that would have made the program slightly slower and more bulky. However, fluid variables make programs much harder to debug and maintain, especially if assignments are made to them in functions other than the ones which establish them. Consequently, we recommend generally avoiding fluid variables. The only reason we used one here is to introduce the concept to issue this advice.

Values assigned at the top-level of muSIMP, outside all function definitions, are called GLOBAL values. Examples are the initial values of muSIMP control variables such as RDS and ECHO, or of muMATH control variables such as PBRCH or PWREXPD. Reference to a global value from

within a function definition is not quite as confusing as reference to a fluid value, and it is indeed onerous to create numerous long lists of parameters in order to pass such environmental control values through a long sequence of function definitions for use deep within.

The property-list technique for set operations is one which we think is more naturally implemented using the Von Neumann programming style. Try to write such a version of UNION: % RDS: FALSE $
% Now, using either style, write an INTERSECTION function using the property-list technique: % RDS: FALSE $
% Taking the FIRST and/or REST of an atom is generally not necessary, but it does in fact have a well-defined value. The FIRST cell of an atom points to the atom's value, while the REST cell points to the property list associated with the atom. For example: %

```
WEATHER: 'FOUL $
PUT ('WEATHER, 'TEMPERATURE, -3) $
PUT ('WEATHER, 'WIND, '((NORTH . WEST), 30)) $
FIRST (WEATHER) &
REST ('WEATHER) &
```

% Integer atoms also have FIRST and REST cells. The FIRST cell of an integer normally points to the integer itself. The REST cell is used to determine the sign of the number: if FALSE the integer is non-negative, if TRUE the integer is negative. %

```
FIRST (7) &
REST (7) &
NINE: 9 $
PUT (NINE, 'TESTING, '(1, 2, 3)) &
GET (NINE, 'TESTING) &
GET (9, 'TESTING) &
```

% All muSIMP data objects (i.e. nodes, names, and integers) have a FIRST cell and a REST cell which can only point to valid muSIMP data objects. Thus, misuse of these selectors cannot accidently give access to non-data objects such as machine language code, stack, print names, etc. This closed pointer universe guarantees the integrity of muSIMP from possible excursions into the unknown.

It is common practice to use EMPTY to test for the end condition as a function proceeds down a list. If such a function is inadvertently given a non-list (i.e. a Non-FALSE atom or a structure whose final REST cell points to a Non-FALSE atom), the function will use the FIRST cell of that atom (i.e. its Value cell) as an element of the list and the REST cell of the atom (i.e. its Property List cell) as the REST of the list. Generally the Property List is a well defined list so the EMPTY test will ultimately cause termination with no ill affects.

We prefer to have non-list arguments give more predictable results confined to the argument. Thus, our internal implementations of MEMBER, REVERSE, and any other functions ordinarily applied to lists use ATOM rather than EMPTY as the termination test. This is slightly faster too, so you may wish to generally avoid EMPTY in favor of ATOM. Alternatively, you can redefine EMPTY to print and return an error message when given a nonFALSE atom: %

```
FUNCTION EMPTY (LIS),
  WHEN ATOM (LIS),
    WHEN LIS EQ FALSE EXIT,
    PRINT ("*** Warning: EMPTY given nonlist ")  EXIT
ENDFUN $
EMPTY (5) $
```
% This is our first example illustrating the fact that conditional exits can be nested arbitrarily deep. The same is true of loops or blocks. This example also illustrates the PRINT function, which prints its one argument the same way that expressions terminated with an ampersand are printed. There is an analogous function named PRTMATH which prints its one argument the same way that expressions terminated with a semicolon are printed.

When functions are called with fewer actual arguments than the function has formal arguments, the remaining formal arguments are assigned the value FALSE. This provides a convenient mechanism for automatically inserting default values for these extra arguments. When an argument evaluates to FALSE, the function can assign the appropriate default value. For example, if the user omits the drive as the third argument of RDS, that function uses the currently logged in drive (i.e. the drive indicated by the last operating system prompt given before entering muSIMP).

There are instances where it is desirable to permit a function to have an arbitrary number of arguments. This is accomplished by making the formal parameter list of a function definition be an atom or non-list rather than a list. The arguments are passed to the function as a single list of argument values, from which the function can extract the values. For example, it is convenient to have a function named MAX which returns the largest of one or more argument values. We can implement this as follows: %

```
FUNCTION MAX ARGLIS,
  MAXAUX (FIRST(ARGLIS), REST(ARGLIS))
ENDFUN $
FUNCTION MAXAUX (BIGGEST, UNTRIED),
  WHEN EMPTY (UNTRIED),  BIGGEST  EXIT,
  WHEN BIGGEST > FIRST(UNTRIED), MAXAUX (BIGGEST, REST(UNTRIED)) EXIT,
  MAXAUX (FIRST(UNTRIED), REST(UNTRIED))
ENDFUN $
MAX (7) ;
MAX (3, 8, -2) ;
```

% This collection of arguments into a list is called NOSPREAD, to distinguish from the SPREAD brand of peanut butter.

Now, suppose that for some reason we already have a list of integers such as %

```
NUMBLIS: '(18, 3, 7, 91, 12, 2) $
```
% and we want to find their maximum. The expression MAX (NUMBLIS) will not work, because MAX is designed for numeric arguments, not for a list of numbers. We could of course extract the elements and feed them

individually to MAX, but this is awkward, especially if we are referring to MAX inside a function and we do not know ahead of time how many integers are in NUMBLIS. Fortunately there is a convenient function named APPLY, which applies the function whose name is the value of its first argument to the argument list which is the value of its second argument. Consequently, we need merely write %

```
APPLY ('MAX, NUMBLIS) &
```

% APPLY works on either SPREAD or NOSPREAD functions. Why don't you try out a few examples: %

```
RDS: FALSE $
```

% A function written in muSIMP is stored internally in a very compact form called D-code (see Section 13.9 of the muMATH Reference Manual). In order to retrieve the definition for use as data, the function GETD (GET Definition) of one argument can be used to decompile the definition and return it is as a linked list. If GETD is given the name of a primitively defined machine language routine instead, the physical memory address of the routine is returned. Finally, if its argument is not a defined function, GETD returns returns FALSE. The following examples show the result of all three types of arguments: %

```
GETD ('UNION) & GETD ('FIRST) & GETD ('FOO) &
```

% Since function definitions can be converted into lists and then recompiled back into D-code, a muSIMP program can actuately be made to modify muSIMP functions! In fact, this is exactly what the TRACE and UNTRACE commands do to the traced function. Other examples which could use this feature include a muSIMP function editor and pretty printer, a cross reference program, and even a compiler all of which could be written in muSIMP.

This is the end of programming lesson 5. These lessons should have provided you with sufficient knowledge to be able to use the muSIMP Section of the Reference Manual to achieve any desired muSIMP programming goal. %

```
ECHO: FALSE $
RDS () $
```

17. GLOSSARY OF PERSONAL COMPUTER TERMINOLOGY

Acoustic coupler: a device for encrypting and decrypting ASCII code in order to transmit it via phone lines to a remote destination.

Address typing: the type of a data object is determined merely from its location or address in memory. Thus special tag bits are not necessary for this purpose.

Argument: a variable whose value a function uses to compute a result. When a function is called, the arguments of the function are listed in parentheses following the function name.

Artificial Intelligence (AI): systems which attempt to make computers perform functions which are normally done by humans such as reasoning, learning, and creative acts.

ASCII (American Standard Code for Information Interchange): this code is the widely used standard for transmitting and storing character data by computers. Each letter, numeral, or special character is represented by a unique number (i.e. 7 binary bits).

Assembly language: the low-level computer language for a processor, each statement of which corresponds one-for-one to a machine language instruction of the processor. Assembly language is used to write high-level languages such as muSIMP since maximum efficiency can be achieved through its use.

Audio-cassette tape: a mass storage medium using the cassette tape commonly used for audio recordings. Although a relatively large amount of data can be stored in this fashion, the access time to a given block of data is very slow.

BAUD Rate: the transmission rate, in bits per second, of data between a computer and a peripheral device such as a terminal or printer. Both computer and peripheral must be set to the same rate to permit communication.

Binary: a condition in which there are but two alternatives. Thus the binary number system (i.e. base 2) has but two values for its digits, either 0 or 1.

Bit (BInary digiT): binary digits are the smallest unit of computer data. They can have as value either 0 or 1. However, collections of bits can represent arbitrarily large and complex numbers and data.

Boolean logic: a binary system of logic named for George Boole. It includes operators such as AND, OR, and NOT which return a truth value based upon the truth values of their operands. Thus boolean operators are widely used in programming to combine logical values and make decisions.

Bootstrap loader: a program usually in ROM which is designed to load

the operating system and then turn control over to the freshly loaded system. Initiating the loader on a computer may either be automatic when the system is turned on or reset, or it may be necessary to perform a number of actions manually.

Byte: a unit of data made of 8 bits. The meaning of the information stored depends entirely on the way in which the byte is interpreted. For instance, a byte can store one ASCII character or it can store an integer in the range from 0 to 256.

Control character: a non-printable ASCII character code used primarily to communicate control directives to the receiving device. Examples include the carriage return, linefeed, and bell characters.

CRT (Cathode Ray Tube): a terminal which uses an electronic vacuum tube to display characters and/or graphics.

Cursor: a block of light on a CRT terminal which indicates your current position on the screen. Its position can be controlled by special control keys on the terminal.

Environment: the current state of a program in the computer. Depending on the language it can include the value and/or properties of all variables and the current definition of all functions.

File: A block of data stored on the mass storage device under a given name. It consists of a collection of records treated as a unit.

Floppy disks: flexible plastic disks coated with a magnetic oxide film which are capable of storing information. They are widely used in conjunction with floppy disk drives as the means of mass storage for a number of popular disk operating systems.

Function: a mapping from a set of arguments to a resulting value. In programming languages functions can have additional side effects which permanently affect the environment.

Global variable: a variable whose value has not been assigned by its use in the formal argument list of a function. Any change to such a variable is recognized by all functions which use it.

Hard disks: rigid disks covered with a magnetic oxide film capable of storing large amounts of data (tens of millions of bytes). Since the disks rotate at a high rate of speed, hard disk drives can access data in a very short amount of time. Currently hard disk systems are more expensive and hence less common than floppy disk storage systems.

Hardware, computer: the physical material making up a computer system. It includes the power supply, integrated logic circuitry, and the container or mainframe in which it resides.

Head alignment: in order for a disk drive to read or write information from or to a given track of a floppy disk, the head must be

accurately positioned over the track. If this is not the case and disk errors are occurring, an alignment must be performed.

Kilobyte: 1024 bytes of storage. This number is used as the standard unit for measuring computer memory sizes since it is an even power of 2.

Local variable: a variable given in a function's argument list which is local to that function. Local variables can be modified within that function without having any effect outside the function. This is a tremendous advantage since it frees the programmer from having to concern him/her self with such global effects.

Login: the procedure for gaining access to a computer. Usually it is not required for personal computers.

Lower-case: the letters of the alphabet expressed as non-capitals.

Mass storage device: the permanent storage of computer information is done on a mass storage device. Although not as fast as RAM, it is capable of storing large amounts of information and the data, for the most part, is immune to computer malfunctions.

Mega-hertz (MHz): a unit of frequency equal to a million cycles per second.

Monitor: a small supervisor program, usually implemented in ROM. It provides a primitive means of communicating with the computer to allow such operations as examining and changing memory, etc.

Mother-board: the individual circuit boards in a personal computer are electrically connected by means of a mother-board. It contains edge connectors in which the circuit boards are inserted.

Non-volatile memory: computer memory which retains its information even after the computer has been turned off. Thus it provides safer, more permanent storage than volatile memory.

On-line: information which can be directly read by a computer. Thus data on a mass storage device is on-line whereas printed matter is off-line.

Operating System: a complete system which provides the basis for user interaction with a computer. It usually provides facilities for the generation, testing, and running of programs and the capability for permanently storing such programs for future use.

Paper tape: a mass storage medium using rolls of paper tape in which the data is stored as a pattern of holes.

RAM (Random Access Memory): the primary means of fast-access read-write data storage within microcomputers. Data stored in RAM is accessed by specifying the address of the data. As distinct from sequential access memory, the time for such an access is independent of the location of the data.

Rational arithmetic: has for its domain or area of applicability all numbers which can be expressed as the ratio of two integers. Thus it includes integer arithmetic as a subset.

Record: a block of data from a disk file which is stored as a unit on a mass storage device. For CP/M like systems, a record consists of 128 bytes.

Reset: to return a computer to the initial condition it was in when first turned on. If a computer program goes wild, reset the computer to initiate the bootstrap loader process. A reset is usually initiated by depressing a switch or button on the computer.

ROM (Read Only Memory): this form of data storage can only be read from, but not changed, by the computer. However, it has an advantage over most RAM's in that it is a non-volatile memory. Thus, it is commonly used to store bootstrap loader programs. ROM is random access memory; however, it is not called RAM since that term is reserved for the volatile, read-write memory.

Sector: a pie shaped area on the surface of a floppy or hard disk. That portion of a track which is contained in one sector is capable of storing one record of data.

Software, computer: the program or set of instructions which specify how a computer is to accomplish a given task. This is considered soft since the same computer hardware can execute many different programs.

Stack, push-down: a data structure used to temporarily store information. The store command is called a PUSH and a fetch command is called a POP. A POP instruction causes the most recently pushed information to be taken from the top of the stack and returned. This is a highly structured and potentially infinite means of storing information.

S-100 Bus: the standard definition of each electrical signal transmitted through the 100 electrical lines on the mother-board of S-100 computers. Thus any S-100 circuit board is theoretically compatible with any computer using the bus.

Track: one of many concentric circles around a disk in which data is stored. Each circle is divided into sectors, one for each record.

Truth value: either the value TRUE or FALSE. Truth values are the subject of Boolean Logic and play a fundamental role in program control.

Variable: a name which is used by a computer program to store a value. The value can be explicitly made using an assignment statement or implicitly made when a function call assigns values to the function's local variables.

Upper-case: the letters of the alphabet expressed in capital letters.

18. INDEX

Function and Variable Name Index

The following is an index of all the important function, variable, and constant names in both muSIMP and muMATH. Each name is followed by a descriptor indicating the name's use, the package in which the name is defined, and the page on which it is documented. Function names are indicated by a set of parentheses following the name which contains the usual number of arguments given to the function.

* Functions defined only for the **APPLE][** version of muSIMP.

```
A>TYPE D:MUMATH.DOC

                MUMATH System Files on the Front and Back
                                of this Disk


    ARITH.SYS ------------    ARITH.MUS

    ALGBRA.SYS -----------    ARITH.MUS
                              ALGEBRA.ARI

    MATALG.SYS -----------    ARITH.MUS
                              ALGEBRA.ARI
                              ARRAY.ARI
                              MATRIX.ARR

    SLVEQN.SYS -----------    ARITH.MUS
                              ALGEBRA.ARI
                              EQN.ALG
                              TRGPOS.ALG
                              TRGNEG.ALG
                              LOG.ALG
                              SOLVE.EQN

    DIFCAL.SYS -----------    ARITH.MUS
                              ALGEBRA.ARI
                              TRGPOS.ALG
                              TRGNEG.ALG
                              DIF.ALG
                              LOG.ALG

    INTCL0.SYS -----------    DIFCAL.SYS
                              INT.DIF

    INTCL1.SYS -----------    INTCL0.SYS
                              INTMORE.INT

    INTCL2.SYS -----------    INTCL1.SYS
                              LIM.DIF

    TAYLOR.SYS -----------    DIFCAL.SYS
                              TAYLOR.DIF

    SUMPRD.SYS -----------    ALGEBRA.SYS
                              TRGPOS.DIF
                              TRGNEG.DIF
                              LOG.ALG
                              SIGMA.ALG
                              LIM.DIF


A>
```